COMPILE-TIME INFORMATION IN SOFTWARE COMPONENTS

by

Scott Owens

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2007

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Scott Owens

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Co-chair:	Matthew Flatt
	Co-chair:	Konrad Slind
		Gary Lindstrom
		John Regehr
		Shriram Krishnamurthi

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Scott Owens in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date	Matthew Flatt Co-chair, Supervisory Committee
Date	Konrad Slind Co-chair, Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Component programming techniques help programmers manage the intrinsic complexity of large software systems by supporting their modular construction from collections of smaller, independent pieces. Each component presents to the other components an abstraction of its functionality with an explicitly specified interface. This dissertation presents two component systems based on the *unit* model of components, and it demonstrates how both systems support flexible placement of component-based abstraction boundaries by allowing component authors to specify compile-time information in interface specifications. It also emphasizes the importance of supporting a simple module system, alongside the component system, for the management of source-code namespaces and compilation.

The first component system is embedded in a model of an ML-like typed functional programming language that includes a Haskell-style module construct. I formalize a type system that supports more flexible specification of type imports and exports than previous *unit* models in similar settings. These specifications are similar to those found in the imports and exports of ML's functors, and I investigate the connections between units and functors, showing formally how a simplified model of functors can be translated into the component system.

The second component system is for languages that can be extended through the use of macros, and I describe its implementation for the PLT Scheme dialect of the Scheme programming language. By allowing definitions of macros to appear inside of component interface definitions, I preserve both compile-time expansion of macros and component independence while allowing different components to encapsulate features that share compile-time information. To accommodate new forms of compile-time information that can accompany new language extensions, the specification language for component interfaces can itself be extended with macros.

To all of my teachers, starting with my parents

CONTENTS

LIST OF TABLES

ACKNOWLEDGEMENTS

I thank my advisors Matthew Flatt and Konrad Slind for their advice and encouragement during the creation of this dissertation, and also for their dedication in helping me arrive at this topic alongside their tolerance as I explored many other potential research areas (e.g., mechanized theorem proving). I also greatly appreciate their efforts in securing funding for my research.

I thank my entire committee for their time and comments, and also for making it far easier for me to schedule the various meetings and defense than fabled.

I am grateful for the efforts of the PLT Scheme team in creating mzscheme and DrScheme, which has inspired and facilitated much of my research. I also thank Ryan Culpepper, my collaborator on the GPCE 2005 paper that was a precursor to the fourth chapter of this document.

I'm especially grateful to my wife Kathy for always being ready to listen when I needed to talk through a rough part of this document.

Lastly, thank you to the owners, staff and patrons of Game Night Games in Salt Lake for all the fun we had when I would take a break from creating this document.

CHAPTER 1

INTRODUCTION

While working on a large program, a programmer should ideally implement, maintain, and reason about only small pieces of it at a time. To allow the programmer to limit his or her focus, the program's high-level structure is a set of modules or components whose boundaries encode useful abstractions. This style of development, called "programming-in-the-large" [DeRemer and Kron 1975], relies on the flexibility to place module or component boundaries wherever the programmer sees a need. Flexible boundary placement depends on the ability of a wide range of different kinds of information to cross the boundaries.

The boundary around a module is described by an *interface* that specifies what the module requires from the rest of the program and what the module provides to the rest of the program. To use the module in a program, the programmer uses a *linking language* to specify how the module's requirements are met by the other modules in the program, and how this module satisfies their requirements. Thus, the crucial features of a module system are the kind of interface definitions and linking languages that it supports.

1.1 Component Programming

A *component* is a kind of module that emphasizes flexible reuse [McIlroy 1969; Szyperski 1997]. Flexibility comes from a loose coupling between the components that constitute an entire program, which the guiding principle behind component systems guarantees:

A component can be implemented, compiled, and deployed independently of other components that might link with it.

This principle requires that a component be a stand-alone entity (with respect to other components) that can be shipped to another developer who can use it, consistent with its interface, without knowledge of the encapsulated details of its implementation. This allows developers to acquire components from diverse sources and use them together.

Not all module systems support components. If the contents of a module $\mathcal{M}_1$ refer directly to another module $\mathcal{M}_2$, then the module $\mathcal{M}_1$ cannot be deployed or even compiled apart from the referenced module $\mathcal{M}_2$. For example, to type check $\mathcal{M}_1$, the compiler will need to consult the definition of $\mathcal{M}_2$ to identify the types of the bindings (or the shapes of the classes, or the definitions of the macros) that $\mathcal{M}_1$ references from $\mathcal{M}_2$. I call these kinds of direct intermodule references *internal linkages* or *concrete linkages* to emphasize that the linkage between the two modules is part of one of the modules itself. Thus, internally linked modules are not suitably independent to be called components.

To achieve independence, the references between components must be independent of the components' definitions. When considering a component's contents, any information about its environment must come from its interface. For example, to type check a component, the compiler consults the component's interface to extract the type of an imported binding (or the shape of an imported class, or the definition of an imported macro). Because a component has a definite meaning apart from any linkages, I call the linkages between components *external linkages.* Thus, the interface attached to a component specifies the assumptions that it can make about its environment once deployed, and the interface also specifies which obligations the component must meet. Similarly, when considering a linking expression, any information about the linked components' contents must also come from their interfaces, not from their implementations.

The expressiveness of a component system is strongly influenced by the kinds of entities that can appear in interfaces when defining and linking components; these are exactly the kinds of entities that a component can abstract over. A large selection of potential abstractions allows a component's author greater flexibility in creating component boundaries, since the boundary can only involve features that can be mentioned in the component's interface. In addition to a flexible interface language, a component system needs a flexible linking language that supports the kinds of abstractions that can appear in interfaces, and that avoids restricting the set of potential linkages further than required by the linked components' interfaces.

Of the wide variety of possible component systems, I consider those whose interfaces comprise a list of *imported* and *exported* source-code resources (functions, values, types, etc.).[1] In such a system, a linking expression must ensure that every import of a linked

[1]Computational resources (CPU time, memory, bus access, etc.) are all potential component interface entities that I do not consider.

component is connected to an appropriate export of a linked component. It is crucial for the system to support linking expressions whose sets of components have cycles in the import and export connections. Otherwise, the programmer will be unable to create a component that encapsulates only part of any mutually recursive program feature.

1.2 Component Compilation

Independent (also called *compositional*) compilation is the crucial characteristic of a component system. If a component can be compiled with knowledge of only its contents and interface, then a compiled version of the component can be used by another developer in any context that satisfies the component's interface. Although the interface must accompany the deployment, the source code need not. To ensure that a component can be compiled in such isolation, its imports must contain all of the information needed to give the component's body a well-defined meaning.

In contrast to independent compilation, many module systems support a notion of *separate compilation* that allows some intermodule dependencies at compile time. A separately-compiled module system allows internal linkages between modules. An internal link allows a module to refer to a specific export of another specific module, thereby preventing the first module from being deployed or used without the referenced module present. Thus, separate compilation allows the semantic meaning of a module to depend on the other modules that it references. Separate compilation differs from whole-program compilation because a module can be compiled apart from the modules that refer to it.

For both internal and external linkages there are two kinds of information that traverse the module/component boundary: run-time information and compile-time information. The import of compile-time information into a module or component must occur when the module or component is compiled, whereas the run-time information is not yet needed. A typical example of compile-time information is the type associated with an imported function, whereas the function's actual definition is run-time information.

Although internal linkages naturally propagate compile-time information across module boundaries, an independent component, while being compiled, does not have access to the other components that will eventually be linked into its imports. Instead, the component's interface must specify all of the compile-time information needed for each of its imports. This document explores the thesis that:

Component interfaces that can include compile-time information enable flex-

ible program partitioning in a typed or extensible programming language.

To this end, I develop two variations on the *unit* system of components [Flatt and Felleisen 1998; Flatt 2006]:

- a component system for a typed language (in the style of ML) where component interfaces can contain type definitions. This system improves the previous model of typed units by supporting translucent type exports in the style of SML's functors [Milner et al. 1997]. It also fully integrates with an internally-linked module system, whereas the previous unit work proposed units as the sole program organization mechanism.
- a component system for an extensible language (the PLT dialect of the Scheme programming language [Kelsey et al. 1998; Flatt 2006]) where component interfaces can be extended to contain new kinds of compile-time information as the underlying language is extended. This system improves the previous implementation of units in Scheme by allowing interfaces to contain compile-time information for language constructs that are not known to the unit system's implementation (i.e., macro-based language extensions).

1.3 Units

The *unit* system closely follows the notion of components explained in Section 1.1; thus, a unit is an externally-linked component that has an explicitly specified interface. The interface comprises a sequence of imported bindings and a sequence of exported bindings, and each binding might have some accompanying information (such as the binding's type). There are two ways of constructing units: *atomic unit* expressions and *compound unit* expressions.

An *atomic unit* contains, in addition to its interface specification, a sequence of definitions, called the unit's body, that create the functionality exported by the unit. The body can refer to imported bindings, and so the imported bindings must contain enough information to compile the body without knowing which component will be used to satisfy the imports. In a simple untyped setting, the binding's name is enough information, and in a typed setting, each import binding must have its type attached.

Units are linked together externally by a *compound unit.* A compound unit's interface is specified as a list of imported and exported bindings, just like an atomic unit's interface. Instead of containing a body of definitions, a compound unit contains a sequence of references to other units that it links together by specifying how the exports of the listed units are used to satisfy the imports of the listed units. Each import of a linked unit is connected to an export of a linked unit, or to an import of the compound unit, and each export from the compound unit must be exported from one of its constituent units.

Because compound unit linking is external, a compound unit expression does not have any information about the contents of the units it links together, except for their interfaces. It cannot even determine whether the linked units are atomic or compound. Link checking ensures that every import is satisfied, and that any compile-time information attached to the import is consistent. For example, if a unit imports a binding x that has type t, then the export being linked into x must have a type compatible with t. No other checks have to be performed; in particular the linkages can be cyclic, and a unit's export can even be used to satisfy its own import.

A fully-linked component program is represented by a unit with no imports. Such a unit can be *invoked*, which descends into the bodies of the linked units and executes their definitions. This process is applied recursively when a linked unit is a compound unit. If the same unit is encountered multiple times, its encapsulated state is replicated and each invocation is completely separate.

Replication prevents unwanted interference when two compound units $\mathcal{C}_1$ and $\mathcal{C}_2$ are linked together, if both of these units contain a common third unit $\mathcal{U}_3$. This is crucial to independence because the programmer linking $\mathcal{C}_1$ and $\mathcal{C}_2$ may not have knowledge of how they were created and which units they contain. When state sharing is desired, it can be arranged explicitly by having $\mathcal{C}_1$ and $\mathcal{C}_2$ both import the interface of $\mathcal{U}_3$. Now the interfaces of $\mathcal{C}_1$ and $\mathcal{C}_2$ indicate which stateful module they need, and the programmer can link them with either one or two copies of $\mathcal{U}_3$ as desired.

1.3.1 Units and Modules

In order to link a unit $\mathcal{U}$ with other units, a compound unit expression must contain a reference to $\mathcal{U}$. Thus, the language of compound unit expressions must have a mechanism for resolving references to unit bindings. A simple solution, such as a single file that contains unit definitions and associates them with global names, is unsuitable because $\mathcal{U}$ might have come from an entirely different source than the units it is being linked with. Thus, the binding mechanism must be able to resolve references to definitions that exist in multiple locations. Although units can perform this sort of binding resolution with their external linkages, attempting to use units to organize unit linking merely raise the same issue in deciding how to organize the organizing units.

An internally linked module system is an excellent way to manage external linking expressions. If all of the units are deployed inside of separate modules, a module containing a compound unit expression can use internal linkages to refer to the desired units. For example, if two modules, $\mathcal{M}_1$ and $\mathcal{M}_2$, both contain a definition for a unit named $\mathcal{U}$ then a compound unit expression that needs a $\mathcal{U}$ can select which one it wants by referring to either to $\mathcal{M}_1$ or $\mathcal{M}_2$. The internally specified nature of the linkage means that the author of the compound expression has complete control over which units to link together.

Internal linkages can be useful in other situations, when external linking is undesirable or impossible, but separation and namespace management are still required. Building a program designed for reuse can take a considerable amount of effort that may not be worthwhile in certain situations. For example, most programs rely on the standard library that comes with their implementation language. Although in certain specialized situations it can be valuable to parameterize a program over the standard library, usually it is not worth the programmer's effort to account for the possibility. Alternately, a programmer might want to rely on a module's behavior that cannot be documented in its interface, and so he does not want to permit the flexibility to link with other components that might not prove fit. The ability to make internal links is therefore an important pragmatic feature beyond the necessity of component-management tasks. To this end, the internally linked module system is not limited to managing unit definitions, but it can manage the other kinds of program definitions as well.

A practical component-oriented language is a two-level system with internally linked modules at the top level, managing component definitions and ordinary definitions alike.

1.3.2 Unit Signatures

Unit interfaces comprise lists of imported and exported bindings, with associated information. It can become tedious to specify such interfaces directly, especially because each piece of an interface must be used on at least one unit that imports it and on at least one unit that exports it. To streamline the process of writing unit interfaces, pieces of an interface can be defined alone in a *signature*. Atomic and compound units specify their interfaces not as a collection of bindings, but as a collection of signatures, each of which can be used in the imports and exports of many different units.

Besides providing a significant convenience, signatures serve as a point of documentation: the intended purpose of a signature's part in an interface can be documented alongside the signature. By using the signature in a unit's interface, a programmer expresses the intention that the unit expects its interface to follow the documentation. Signatures also facilitate reuse because programmers can write components that import or export widely distributed signatures, and thereby gain the ability to easily link with other units written to those signatures.

Because a unit's imports and exports are needed to compile the unit, its signatures must be attached to it at compile time. An internally linked module system provides an ideal way to attach them since the links can be followed by the compiler.

Flatt's and Felleisen's unit system does not include a separate signature construct, but Flatt's implementation of units for PLT Scheme does. My typed system of units does not include an explicit treatment of signatures, but my Scheme system does.

1.3.3 Units in a Typed Language

In a component-oriented typed programming language, types are the key compile-time information that must appear in component interfaces. Units can support a typed language by requiring types to appear as annotations on imported and exported bindings, so that the compiler can type check unit bodies and unit linkages independently of one another. Furthermore, units allow types themselves to be imported and exported, and the types of imported and exported values can use the imported and exported types.

A type can be imported and exported from a unit in two ways: opaquely or translucently. An *opaque* (also called *abstract*) type hides all information about its definition. An opaque type export from a unit is similar to an existential type; the unit exports the information that the type exists and nothing else. An opaque type imported into a unit is likewise similar to a universal type; the unit's body must be able to handle any type that could be given to the import.

When an opaque type t is used in conjunction with values whose types involve t, it enforces abstractions along the unit's boundary. For example, consider a signature with t and two functions $\mathit{make\text{-}t}$ and $\mathit{print\text{-}t}$ where $\mathit{make\text{-}t}$ has type $(\mathbf{int} \rightarrow t)$, and $\mathit{print\text{-}t}$ has type $(t \rightarrow \mathbf{string})$. If a unit imports this signature, it can create values of type t from **int**s, and it can then print them to strings, but it cannot perform any operation that depends on how t is defined. If a unit exports this signature, it can define t as any type it chooses because it is certain that no external code can rely on t's implementation.

This allows different units that export the signature to choose different types in their implementation.

A *translucent* (also called *transparent* and *manifest*) type exposes its definition. A unit importing a translucent type can take advantage of knowledge of the type's structure, and a unit exporting a translucent type is required to use compatible types internally. Similarly, in linkages, any type linked into a translucent type must be compatible. In contrast, any type, translucent or opaque, can be linked into an opaque type import. Translucent types can be combined with opaque types to partially hide a type's implementation. For example, an interface could include an opaque type t_1 and a translucent type t_2 that is equal to the tuple type $< \textbf{int} \times t_1 >$. An importing unit could rely on values of type t_2 being tuples with integers in the first position, but would not know anything about the value in the second position. Similarly, an exporting unit would have to define t_2 as a tuple of an integer and some other type of its choice.

Translucent and opaque types have been well-studied in the context of ML's functors [Harper and Lillibridge 1994; Leroy 1994], and their combination is crucial to resolving well-known problems such as the "diamond import" problem (discussed further in Chapter 2). However, functors are not components as defined in Section 1.1 because they do not support cyclic linking patterns. In contrast, the previous model of typed units supported cyclic linking and opaque types with a limited facility for type abbreviations, but it did not support translucent types that could exhibit full interactions with the opaque types.

1.3.4 Units in an Extensible Language

Scheme is an *extensible* programming language; new constructs can be added to the language without modifying its core implementation. A language extension can introduce new kinds of compile-time information that are required for the compilation of the new construct.

For example, Scheme can be extended with syntax to create algebraic datatypes and with pattern-matching expressions to deconstruct them. To process a pattern-matching construct over a dataype $\mathcal{D}$, the pattern-matching extension needs to know $\mathcal{D}$'s layout, so that it can generate code that extracts values from constructed instances. For a unit to encapsulate a pattern-matching expression apart from the datatype definition, the compile-time information about the layout of the datatype needs to appear in the unit's signature. Since the information was created by a language extension, the language of unit signatures also needs to support extensions to allow it to include new kinds of compile-time information.

1.4 Contributions and Evaluation

This dissertation proposes novel solutions to four problems encountered with the unit component system. First, it addresses a significant inflexibility in the handling of type information by adding full support for translucent type imports and exports to unit interfaces. Second, it integrates an internally linked module system with the unit system in a typed setting. Third, it removes the need to extend the core unit implementation to account for language extensions by adding an extensible facility for supporting new language constructs, including those that contain compile-time information. Fourth, it alleviates several practical inconveniences in using units by adopting a combination of nominal and structural signature matching and by supporting linkage inference for compound units.

I prove my thesis by demonstrating the ability of my component systems to express programs whose natural component-oriented decomposition requires certain compile-time information to travel across the components' boundaries. In particular, the applicability of component-based programming to these examples is made possible only by my extensions to units.

1.5 Related Module Systems and Component Systems

Many programming languages use an internally linked module system to support a hierarchical name space for program definitions, and often to manage multifile programs. Table 1.1 presents an overview of the module systems of several languages. In contrast, few languages support a component system. In the languages that lack component systems, strict adherence to internal linking can prove excessively inflexible. This leads many internally-linked module systems to allow some amount of external resolution of internal linkages, yielding a hybrid linking solution that has neither fully general external linking, nor fully concrete internal linking. It is a primary tenet of this dissertation that the two types of linking are orthogonal, and that they should be implemented with separate language constructs.

Table 1.1. Module systems overview. The "Dot Notation" column indicates individual internal links that have both the module name and definition name (e.g., `modName.defName`). The "Import Notation" column indicates that a module can incorporate another module's definitions into its namespace to be referenced as local definitions (e.g., `import modName` followed later by `defName`). The "Nested Mods." column indicates that a module can be defined inside of another module. The "Comp. Mgmt." column indicates that the module system allows language-based tools to manage compilation and recompilation of module-based programs. This can only happen when module paths contain enough information to unambiguously locate the referenced module.

Language	Construct	Dot Notation	Import Notation	Nested Mods.	Comp. Mgmt.
PLT Scheme	**module**		✓		✓
Bigloo Scheme [Serrano 2004]	**module**		✓		
Scheme48 [Kelsey et al. 2005]	**structure**		✓		
Chez Scheme [Waddell and Dybvig 1999]	**module**	✓	✓	✓	
Haskell [Jones 2003]	**module**	✓	✓		✓
SML [Milner et al. 1997]	**structure**	✓	✓	✓	
SML CM [Blume and Appel 1999]	**Library**				✓
SML MLBasis [Cejtin et al. 2005]	**basis**	✓	✓	✓	✓
SML extension [Swasey et al. 2006]	**unit**		✓		✓
OCaml [Leroy 2004]	**module**	✓	✓	✓	
Java [Gosling et al. 2000]	**package**	✓	✓		✓[2]
Modula-2 [Wirth 1982]	**MODULE**	✓	✓	✓	

[2]except for the effect of the `CLASSPATH` environment setting.

For example, in Java, a reference to a class through a full package name is resolved relative to the contents of the `CLASSPATH` environment setting. The module that is actually referenced can be changed by changing the `CLASSPATH`; however, the class names and structures are expected to be the same regardless of what the `CLASSPATH` is. In fact, problems can arise if the compile-time `CLASSPATH` is different from the run-time one. Similarly, OCaml treats top-level ML structures as compilation units, and an internal link in one refers to another by name. However, it is up to the invocation of the linker to determine which of potentially many top-level modules with the same name are to be included in the final program.

Of languages with constructs for component-oriented programming, ML has the most in common with units, and will be discussed in further detail in Chapter 2. Object-oriented programming offers some of the benefits of component-oriented programming. In particular, it places a strong emphasis on accessing objects only through explicitly specified interfaces. Because this dissertation concentrates on component interfaces, its techniques can be applied to these sorts of systems, although my focus is limited to units. For example, in a syntactically extensible, object-oriented language, macro definitions should appear in class interfaces, and the addition of a new kind of class member should appear alongside the definition of a corresponding new interface entity.

Recently, the Scala [Odersky and Zenger 2005] programming language has combined several object-oriented technologies, including mixin-based inheritance, to support component abstractions and compositions. In Scala, a component is represented by a class, which can be abstracted over its parent classes and composed via a mixin application that supplies the missing parents. Similar to units, the mixin-based approach relies on the result of a component composition being a component itself. Scala's class-based system also handles recursive linking naturally. A significant difference between the unit and the Scala approach comes from Scala's "class linearization" which reduces the directed, acyclic graph of class parents to a duplicate-free list. This leads to the kind of interference discussed in Section 1.3.

The CORBA [Vinoski 1997] system for dynamic components or object-oriented languages follows the methodology of external linking and independent compilation. A component in CORBA is represented not as a class (as in Scala above), but as an object. Each object can make requests of other objects—essentially remote method calls—whose identities are not determined until run time. As in units, these objects are described by interfaces which the CORBA requests must follow. However, the language for specifying interfaces, OMG IDL, is not extensible, so new kinds of objects and primitive data (perhaps from a new language) may not fit well into a CORBA-based framework. This restriction is important to CORBA's support for multiple nonextensible languages, since interoperability is simplest when the interoperating languages are similar. However, in the setting of a single, extensible language, the interface language must be extensible.

Mixin modules [Duggan and Sourelis 1996; Wells and Vestergaard 2000; Hirschowitz et al. 2004; Makholm and Wells 2005] support the separation of mutually recursive program values across modules. Like units, mixin modules separate the composition operation from the invocation operation. Ancona and Zucca [2002] define a core calculus for module systems with higher-order and recursive features, and their line of work has produced a module system for Java [Ancona and Zucca 2001]. More recent work on polymorphic bytecode [Ancona et al. 2005] addresses both direct and indirect references, much like structures and functors. This work is similar in spirit to units, but with no connection to SML-style concerns such as translucency, sharing by specification, etc.

The goal of application product lines is similar to that of component programming: a program is assembled from several pieces, each of which represents some particular feature of the program. In fact, product line system such as AHEAD [Batory et al. 2004] and unit-based systems such as Jiazzi [McDirmid et al. 2001] have similar expressive power on some problems [Lopez-Herrejon et al. 2005]. However, a slight difference in the goal of the two approaches leads to a fundamental technical split. The product line approach focuses on the automatic creation of several similar applications from a repository of in-house source code. For example, a product-line methodology might automate the construction of a restricted student version of a program alongside a fully functional professional version. Because all of the source code is available when the application is constructed, product line techniques do not have the strong requirements for independent compilation and deployment that are fundamental to component-oriented technologies, and that impact design choices throughout this dissertation. In particular, the AHEAD tool approaches feature composition based on program source fragments.

Despite these differences, AHEAD illustrates an important facet of extensibility. It not only composes program source fragments, but other resources such as makefiles and XML metadata. As new resources are required for a particular project, AHEAD must be extended so that it knows how to compose the new resources. A special purpose tool supports the construction of such extensions to AHEAD. This situation arises whenever a component system is given a new type of construct to manage, which is exactly the problem that this dissertation solves for units in an extensible language.

Findler and Flatt [1998] describe idioms for creating extensible software (programs that can be extended with new features, as opposed to extensible languages) that rely on a combination of component- and object-oriented programming techniques. They use PLT Scheme's existing unit system and class system to create architectures where a component can represent either a new data variant or a new operation on a family of existing variants. In both cases, the component imports classes and exports subclasses. Because PLT Scheme's classes are first-class values that are parameterized over their superclass, the unit system provides no special support for these idioms; the body simply defines classes whose superclass expression refers to a unit import. Jiazzi [McDirmid et al. 2001] supports similar idioms for the Java programming language with units whose interfaces include the description, although not implementation, of imported and exported classes to facilitate type checking.

1.6 Outline

Chapter 2 presents a typed language that includes internally linked modules and units that fully support opaque and translucent types. It includes several examples drawn from the literature on functors, and furthermore, it precisely relates units and functors with a formal translation from functors into units. Chapter 3 presents a formal type system and operational semantics for typed units, along with a proof of type-soundness. Chapter 4 describes Scheme's mechanisms for language extensibility, and demonstrates how to incorporate those into unit signatures. Chapter 5 presents extensions to units designed to make component-oriented programming practical. It includes several examples from the DrScheme's [Findler et al. 2002] source code. Chapter 6 presents the implementation of the unit constructs as language extensions to Scheme, and it discusses my experience applying them to the DrScheme source code. The appendix contains a full grammar for the Scheme unit system, which is introduced throughout Chapters 4 and 5.

CHAPTER 2

UNITS AND MODULES IN A TYPED LANGUAGE

Chapter 1 explained several of the requirements that the module system for a typed functional language should meet: it should include an internally linked module construct that can express concrete program organization, and it should include an externally linked component system that supports these features:

- the ability to compile a component without regard to any linkages that it participates in (that is, independent component compilation)
- opaque type imports and exports to hide a type's definition
- translucent type imports and exports to expose a type's definition
- hybrid type imports and exports that combine both translucent and opaque types to partially expose and partially hide a type's definition
- recursive (cyclic) linkages
- multiple independent instantiations of a component.

Besides these, the component system should also support type sharing, the ability of a component to require that several of its imported opaque types are all linked to equivalent types. Sharing is crucial when a component imports from two different components, and needs to use the functionality of one in conjunction with the other. A fully general treatment of translucent types allows sharing to be expressed without the introduction of further concepts or constructs [Harper and Lillibridge 1994; Leroy 1994; Leroy 1996].

Existing typed programming languages typically include module systems that support some variant of internal linking. For example, in Java [Gosling et al. 2000], a program comprises a set of classes, organized into packages. Code inside of one class uses a dot notation to directly link to either a class C (which is compile-time information in Java) inside of a package P or to a (static) variable v inside of the class: $P.C$ and $P.C.v$, respectively.[1] Links between classes can be cyclic, which means that a minimal unit of separate compilation can be a set of classes. Haskell's module system [Jones 2003] is similar to Java's. All type and value definitions occur inside of a module, and modules are directly linked by **import** statements which make one module's values and types available in another. Module linkages can be cyclic (put another way, module definitions can be mutually recursive), so a set of modules can require simultaneous compilation.

Neither Haskell nor Java has a component system that supports external linkages. The Jiazzi [McDirmid et al. 2001] component system extension to Java is based on units that import and export classes. Jiazzi's design conforms to the principles laid out in Chapter 1; a signature in Jiazzi includes the types of the methods and fields of an imported or exported class. The differences between Java and typed functional languages give Jiazzi's units a different flavor from a component system suitable for an ML-like language. In particular, Java's type system is nominal—type equality and subtyping are based only on explicit declarations—and ML's type system is structural—type equality is based on a type's shape. Jiazzi uses Java's object-oriented nominal subtyping (with interfaces) to perform the kinds of information hiding/exposure offered by opaque and translucent types in the structural setting.

The ML [Milner et al. 1997; Leroy 2004] family of languages has a module system that supports internal and external linking with two related constructs: *structures* and *functors*. A structure comprises type and value definitions that can be referenced with dot notation from other structures. Unlike Haskell's modules, structures cannot have cyclic references between them, and furthermore, structures support an operation called *sealing*. A structure is sealed by attaching a signature that specifies the type and value definitions that are accessible from outside of the module. In the case of type definitions, only the information about a type that is given in the signature is visible outside of the structure. The ability to hide type information differentiates sealing from module systems (e.g., Haskell's) that forbid external references to nonexported names without hiding the types of exported values. For example, if a structure is sealed with the signature `S1` then, outside of the structure, nothing is known about the type of `v`.

```
signature S1 = sig type t; val v:t end
signature S2 = sig type t = int; val v:t end
```

[1]An "**import** P" statement is a shorthand that allows the omission of P in the paths.

If signature `S2` is used, then it is known that `v` has type `int`. The type specification for `t` in `S1` is an example of an opaque type, and its specification in `S2` is an example of a translucent type.[2]

A functor is a module-system level function that consumes and produces structures. Its input structure is described by a signature; any structure matching that signature can be used as the argument to the functor in a functor application. Since functor application produces a structure, multiple functors cannot be directly composed. Instead, each is applied to the result of a previously applied functor. Thus, functors and structures are tied closely together by the functor application linking mechanism. Because ML structures do not support cyclic linkages, ML functor linking patterns must be acyclic as well. Each structure produced by a functor application is accessible to other structures through internal links, which means that its contents must be fully specified, even if the structure is only intended to be an intermediate step in a sequence of applications.

Various extensions to ML have proposed facilities for recursive structures and functors [Crary et al. 1999; Russo 2001; Dreyer 2005b]. This chapter can be viewed as a different approach to the same problem of satisfying the above feature list. Instead of adding recursion features to ML's module system, I add translucent types and direct linking to the already recursive system of typed units [Flatt and Felleisen 1998] (although the formal definition of my type system uses the notation and many of the techniques of ML module system formalizations instead of those from Flatt's and Felleisen's formalism). Similar to ML, there is one construct for internal linking and one for external linking: **module** and **unit**, respectively. The coupling between modules and units is looser than between structures and functors; external linkages between units can be resolved incrementally without producing intervening modules. Loose coupling is crucial in allowing units to naturally support compositions that contain cyclic linkages. It also permits the module system to manage separate compilation (not independent compilation) without complicating the component system.

This chapter explains the module/unit separation and relates its expressive power to that of the SML module system. In particular, I offer an alternate semantics for the SML module system with a translation into **module** and **unit**. For SML programmers, this alternate view may offer insights into enabling mutual recursion among SML functors.

[2]Sealing is performed in Standard ML with the `:>` syntax. The `:` syntax does not perform sealing; it fully exposes the types of the structure's values.

For non-SML programmers, the alternate semantics complements the existing semantics of SML and of units to foster a deeper understanding of key module and component system technologies.

Section 2.1 introduces units and modules and shows how they complement each other. Section 2.2 contains several examples of the expressiveness of units. Section 2.3 relates units and modules to functors and structures, and gives a translation from a model of generative structures and functors into modules and units. Section 2.4 discusses the extensions to my model that are necessary for practical programming.

2.1 Modules and Units

Figure 2.1 presents the syntax of the language of modules and units over a simply-typed core language that includes sum and product types, value definitions, and type definitions. The module system comprises the **module** and path constructs, and the unit system comprises the **unit** and **compound** expressions and the **invoke** definition.

The language requires that each value-binding identifier (x^i) be annotated with its type (*Ty*), so that it does not require type inference. The expressions (E) of the language are typical of a simply-typed λ-calculus, with the addition of the unit system. The language has the standard types for integers, sums, products, and functions. Units are supported as first class values, following Flatt and Felleisen's system,[3] so the language also includes a type for units; this type fully contains the type and value bindings (but not value definitions) of the unit's interface to ensure a phase distinction between type checking and execution. A type can also contain a path (P_t) to a type definition.

2.1.1 Modules

A program (*Prog*) is a sequence of modules (M), each of which contains a sequence (*Ds*) of definitions (D). Each definition's scope extends from the immediately following definition to the end of the module (i.e., a module's definitions are not recursively scoped). Definitions are accessed from outside of the module with a path to either a value (P_x) or to a type (P_t). For example, the path $m^1\mathbf{.}n\mathbf{.}t$ refers to the definition of type t inside of module n which is itself defined inside of m^1, a module defined in the scope where the path appears.

[3]With first-class units, the code that links units is expressed in the programming language itself, instead of in a dedicated linking language.

Names, identifiers, and paths:

m, t, x	$=$	module, type, and value names	
n	$=$	$m \mid t \mid x$	*names*
ns	$=$	$\epsilon \mid n\ ns$	*name sequences*
i, j	$=$	$\mathbb{N}$	*stamps*
id	$=$	$m^i \mid t^i \mid x^i$	*identifiers*
ids	$=$	$\epsilon \mid id\ ids$	*identifier sequences*
P_m	$=$	$m^i \mid P_m.m \mid P_m.\widehat{m}$	*paths to modules*
P_x	$=$	$x^i \mid P_m.x \mid P_m.\widehat{x}$	*paths to values*
P_t	$=$	$t^i \mid P_m.t \mid P_m.\widehat{t}$	*paths to types*
P	$=$	$P_m \mid P_t \mid P_x$	*paths*

Types:

Ty	$=$	$\mathbf{int} \mid Ty + Ty \mid Ty \times Ty \mid Ty \rightarrow Ty$	*base, sum, product, and function types*
	$\mid$	P_t	*type references*
	$\mid$	$\mathbf{unitT}\ \mathcal{M}\ (ns \rightarrow ns)$	*types for units*
B	$=$	$x^i{:}Ty$	*value bindings*
	$\mid$	$m^i{:}\mathcal{M}\ (ns)$	*module bindings (with provided names)*
	$\mid$	$t^i \mid t^i{=}Ty$	*opaque and translucent type bindings*
$\mathcal{M}$	$=$	$\epsilon \mid B,\mathcal{M}$	*module, import/export descriptions*
Γ	$=$	$\epsilon \mid \Gamma,B$	*typing contexts*

Terms:

$Prog$	$=$	$M \ldots M$	*programs*
M	$=$	$\mathbf{module}\ m^i\ \mathbf{provide}\ ns = Ds$	*module definitions*
	$\mid$	$\mathbf{module}\ m^i{:}\mathcal{M}\ \mathbf{provide}\ ns = Ds$	*sealed module definitions*
Ds	$=$	$\epsilon \mid D\ Ds$	*definition sequences*
D	$=$	$x^i{:}Ty{=}E \mid t^i{=}Ty \mid M$	*value, type, and module definitions*
	$\mid$	$\mathbf{invoke}\ E\ \mathbf{as}\ m^i{:}\mathcal{M}$	*unit invocation*
E	$=$	$\mathbb{Z}$	*integer constants*
	$\mid$	P_x	*value references*
	$\mid$	$\mathbf{injl}\ E \mid \mathbf{injr}\ E \mid \mathbf{case}\ E\ \mathbf{of}\ E\ +\ E$	*sum expressions (use λ for binding)*
	$\mid$	$(E,E) \mid \pi_1 E \mid \pi_2 E$	*product expressions*
	$\mid$	$\lambda x^i{:}Ty.E \mid E\ E$	*function expressions*
	$\mid$	$\mathbf{let}\ Ds\ \mathbf{in}\ E$	*let expressions*
	$\mid$	$\mathbf{unit\ import}\ \mathcal{M}\ \mathbf{export}\ \mathcal{M}.\ Ds$	*atomic unit expressions*
	$\mid$	$\mathbf{compound\ import}\ \mathcal{M}\ \mathbf{export}\ \mathcal{M}$	
		$\quad \mathbf{link}\ U \ldots U\ \mathbf{where}\ L \ldots L$	*compound unit expression*
U	$=$	$E{:}\mathbf{import}\ \mathcal{M}\ \mathbf{export}\ \mathcal{M}$	*interface annotation*
L	$=$	$id \leftarrow id$	*linking expression*

Figure 2.1. Syntax of typed modules and units

Following Leroy's manifest types approach, identifiers (id) are used for references within the module, and names (n) are used for references into the module from outside. The α-relation can modify only the stamp on the name of a module-level definition; it cannot change the name itself. This restriction prevents α-renaming from interfering with the use of names for external access into a module, while preserving its ability to freely and locally rename identifiers inside of a module.

A path can contain both names and hatted names ($\widehat{x}$, $\widehat{t}$, $\widehat{m}$). A name-based reference into a module is valid only when that name is listed in the module's **provide** clause, but a hatted-name reference ignores the **provide** clause to allow access to unprovided module bindings. Hatted names are prohibited in source programs because they can violate abstraction boundaries; however, the path grammar and type system support them because the type system can require access to unprovided type definitions during type checking (see Section 3.1.1). This access is needed when a module defines and provides a value whose type refers to an unprovided type definition.

A module definition can optionally give an explicit module description ($\mathcal{M}$[4]) that seals the module, similar to how a structure is sealed in ML. The difference between sealing and providing is that each type occurring in the description $\mathcal{M}$ must be defined in $\mathcal{M}$ (or in an enclosing scope), and such definitions may be defined translucently or opaquely. In contrast the **provide** clause merely restricts the ability to mention a type or value by name from outside of the module; it does not permit the creation of opaque types, or restrict the types of provided values.

As the form for managing internal linking, modules have two roles. First, module definitions can be nested for fine-grained management of name grouping and lexical scoping. Second, top-level modules naturally form the boundaries of *separate compilation*, because every dependency on another top-level module is explicit in the body of the module. By restricting a program's top-level to module definitions only, and by requiring an acyclic dependency relation, a single module forms the minimal unit of separate compilation. Units, as first-class program entities alongside functions, products, etc., fit naturally into the management of modules.

[4]I occasionally treat members of Γ as members of $\mathcal{M}$ and members of $\mathcal{M}$ as members of Γ when conventient, since they are both just lists of Bs.

2.1.2 Units

A **unit** expression creates an atomic unit with explicitly described imports and exports ($\mathcal{M}$). The unit's body must define each exported binding (B), and its body can refer to any imported binding. Each imported or exported binding can specify a value, opaque type, translucent type, or module. Module bindings support the grouping of imported and exported values, types, and modules. The type of a unit includes a single context that contains the unit's imported and exported bindings, and it also includes a listing of which of these bindings are imports and which are exports. The single context can interleave imported and exported bindings, allowing the types of imported values to refer to types exported from the unit. This feature helps avoid the "double-vision" problem (see Section 2.3.3). However, neither the unit's body, nor its interface is recursively scoped.

Following modules, the definitions in a unit's body are correlated internally (including in the correspondence of the definitions and uses in a unit's body with the unit's export and import contexts) using identifiers, so the α-relation cannot change their names, only their stamps. This restriction means that a unit's imported and exported names form a concrete and unchangeable part of its interface. External linking expressions rely upon the consequential ability to unambiguously refer to particular imports and exports from a unit with no knowledge of the unit itself.

A **compound** expression links units together to form a new unit. Each linked unit (U) specifies an expression along with the imports and exports expected of the expression. The linkages (L) specify which type, value, and module exports from the linked units are used to satisfy each of the linked units' imports. Each identifier on the left side of a linkage must be listed in one of the import contexts accompanying the linked units, and each identifier on the right of a linkage must be listed in one of the export contexts, or in the compound unit's imports. The compound's **export** clause specifies which of the linked units' exports are themselves exported from the compound unit.

An **invoke** definition evaluates the definitions inside of a unit that has no imports, and it places the exported values and types in the enclosing scope, accessible through the given module identifier. The semantics of **invoke** enforce the phase distinction. Because the type of its argument expression is known before run-time, and because that type contains all of the visible type information about the unit's exports, the types that the **invoke** expression places into the module result are completely known before run-time.

At run-time, the evaluation of an **invoke** definition causes the evaluation of the definitions contained in the argument unit.

The **unit** and **compound** forms are value expressions (E) to support units as first-class values. However, the invocation of a unit does not produce a value; it instead produces definitions that bind names to values, types, and modules. Thus, the **invoke** form is not a value expression, but a definition form (D) that defines a module that groups the produces definitions in a single place.

A subtype relation $<:$ arises naturally from unit values. Roughly speaking, a unit with fewer imports or more exports can be used in place of a unit with more imports and less exports. Section 3.1.2 discusses subtyping in further detail.[5]

2.1.3 Other Expressions

The expressions for constants, products, functions are typical. The constructor expressions for sums are also typical; however, the case expression does not directly bind the contents of the sum value it deconstructs (to avoid the treatment of **case** as a binding construct). Instead **case** passes the sum's contents to one of the function expressions it is given: **injl** contents to the expression immediately before the plus and **injr** contents to the one after.

In this language, the **let** expression is not just a macro over functions, because functions can only bind values as parameters. However, **let**'s binding is an arbitrary definition form, including **module** and **invoke** definitions, allowing the language to support locally-scoped unit invocations (e.g., invoking a unit in a function body).

2.1.4 Example

Figure 2.2 presents the skeleton of a set library, and its use, based on Objective Caml's set library which parameterizes the set over the type of items in the set. The example combines opaque and translucent types to hide the type of the set while exposing the type of elements in the set. Figure 2.2 uses an ML-style syntax to illustrate the basics of unit programming; Figure 2.3 contains the same example written in the syntax of the formal system.

The `order` module provides the signature for ordered types. The `ordered_int` module provides a unit `oi_unit` whose exports conform to `order_sig`, enriched with the informa-

[5]The complications of subtyping in the core language can be avoided by taking units out of the core and placing them in a module level, as is typically done in ML-style functor systems.

```
module order provide order_sig
  signature order_sig = sig
    type t;
    val compare:t->t->int
  end

module ordered_int provide oi_unit =
  val oi_unit =
    unit import () export order.oi_sig where t = int .
      type t=int
      val compare:int->int->int = ...

module set provide set_unit =
  signature set_sig = sig
    type t;
    type elt;
    val add:elt->t->t;
  end
  set_unit =
    unit import I=order.order_sig
         export set_sig where elt = It .
      type elt = It
      type t = ... elt ...
      val add:elt->t->t = ... Icompare ...

module main provide set s =
  val int_set_unit =
    compound import () export set.set_sig where elt = int
      link O=ordered_int.oi_unit
           S=set.set_unit
      where (SIcompare <- Ocompare) (SIt <- Ot)
  invoke int_set_unit as set
  val s = set.add 12 ...
```

Figure 2.2. Set example: ML-style syntax

```
module $\mathit{ordered\_int}^0$ provide $\mathit{oi\_unit}$ =
  $\mathit{oi\_unit}^0$:unitT [$t^0$=int, $\mathit{compare}$:$t^0 \to t^0 \to$int] ( $\to$ $t$ $\mathit{compare}$) =
    unit import [] export [$t^0$=int, $\mathit{compare}$:$t^0 \to t^0 \to$int].
      $t^0$=int
      $\mathit{compare}^0$:int$\to$int$\to$int = ...

module $\mathit{set}^0$ provide $\mathit{set\_unit}$ =
  $\mathit{set\_unit}^0$:unitT [$\mathit{It}^0$, $\mathit{elt}^0$=$\mathit{It}^0$, $t^0$, $\mathit{Icompare}^0$:$\mathit{It}^0 \to \mathit{It}^0 \to$int, $\mathit{add}^0$:$\mathit{elt}^0 \to t^0 \to t^0$]
                    ($\mathit{It}$ $\mathit{Icompare}$ $\to$ $t$ $\mathit{elt}$ $\mathit{add}$) =
    unit import [$\mathit{It}^0$, $\mathit{Icompare}^0$:$\mathit{It}^0 \to \mathit{It}^0 \to$int]
         export [$t^0$, $\mathit{elt}^0$=$\mathit{It}^0$, $\mathit{add}$:$\mathit{elt}^0 \to t^0 \to t^0$].
      $\mathit{elt}^0$ = $\mathit{It}^0$
      $t^0$ = ... $\mathit{elt}^0$ ...
      $\mathit{add}^0$:$\mathit{elt}^0 \to t^0 \to t^0$ = ... $\mathit{Icompare}^0$ ...

module $\mathit{main}^0$ provide $\mathit{set}$ $s$ =
  $\mathit{int\_set\_unit}^0$:unitT [$t^0$, $\mathit{elt}^0$=int, $\mathit{add}^0$:$\mathit{elt}^0 \to t^0 \to t^0$] ( $\to$ $\mathit{elt}$ $t$ $\mathit{add}$) =
    compound import [] export [$t^0$, $\mathit{elt}^0$=int, $\mathit{add}^0$:$\mathit{elt}^0 \to t^0 \to t^0$]
      link $\mathit{ordered\_int}^0$.$\mathit{oi\_unit}$ : import [] export [$t^1$=int, $\mathit{compare}^1$:$t^1 \to t^1 \to$int]
           $\mathit{set}$.$\mathit{set\_unit}^0$ : import [$\mathit{It}^2$, $\mathit{Icompare}^2$:$\mathit{It}^2 \to \mathit{It}^2 \to$int]
                             export [$t^2$, $\mathit{elt}^2$=$\mathit{It}^2$, $\mathit{add}$:$\mathit{elt}^2 \to t^2 \to t^2$]
      where ($\mathit{Icompare}^2 \leftarrow \mathit{compare}^1$) ($\mathit{It}^2 \leftarrow t^1$)
  invoke $\mathit{int\_set\_unit}^0$ as $\mathit{set}^0$:[$t^3$, $\mathit{elt}^3$=int, $\mathit{add}^3$:$\mathit{elt}^3 \to t^3 \to t^3$]
  $s^0$:$\mathit{set}^0$.$t$ = $\mathit{set}^0$.$\mathit{add}$ 12 ...
```

Figure 2.3. Set example: Figure 2.1 syntax tion that the type of ordered elements `t` is, in this case, `int`. The `set` module provides the signature of sets, which includes the type of a set `t`, the type of elements in the set `elt`, and the operations on the set (represented by `add`). It also provides `set_unit` which imports an implementation of ordering, and exports a set based on that ordering. The `I` prefix is used to disambiguate between the two different types `t` in `set_unit`'s interface. Because the type of ordered elements `t` is imported opaquely, the only operation that the set implementation can perform on a value of type `t` is the imported comparison (which the `add` function uses).

Although the implementation of the `add` function in `set_unit` is elided, it could include internal references to standard library functions managed by the module system. If the programmer wishes to support multiple implementations of these library functions, the references would become unit imports instead.

The `main` module links `oi_unit` and `set_unit` into `int_set_unit`, using internal, module system links to concretely identify the two units to link. The compound unit `int_set_unit` exports the set type `t` opaquely, but exports the type `elt` translucently, so that the `add` function can add integers to the set while the set's implementation type is hidden. The `invoke` statement runs `int_set_unit` and groups its exports under the module name `set`. The `S` and `O` prefixes support disambiguation when multiple units share imports or exports, such as the `t` type export from `set_unit` and `oi_unit`.

Figure 2.3 adds in the explicit type annotations and the identifier marking superscripts; because unit expressions have explicit import and export clauses, the **unitT** annotations can be directly taken from those clauses. It also removes the signature notation by copying the signatures into the unit interfaces directly; the `where` equations have been inlined into the interfaces as translucent types, and the unit import prefixing notation (`I`) has been resolved by adding the prefix into the names.

The import and export clauses listed with the expressions in the **link** clause serve the same purpose as the `O` and `S` prefixes. Each of these is matched against the corresponding unit by name, so that $Icompare^1$ and $Icompare^2$ match. The **where** clause treats the imports and exports as binding by identifier, and can thus refer to either t^1 or t^2 without ambiguity. (Since linking can be recursive, the **where** clause is allowed to connect any export into any import, of a compatible type, which means that the ordering of the **link** clause cannot be used to resolve **where** linkages.)

2.2 Expressiveness

Units can express several basic language constructs, including parametric polymorphism, recursive functions, and recursive datatypes. Along with the set example (Figures 2.2 and 2.3), the expressiveness examples of this section illustrate the basic methods that units use to support recursive linking and type abstraction. (This section freely omits type annotations and identifier superscripts where they are uninteresting).

A polymorphic function can be encoded as a unit that imports the relevant type variables and exports the function. Figure 2.4 contains such a definition of the polymorphic function composition combinator, *o*. The unit *o_unit* defines the function, and the units *types_unit* and *c* instantiate it to particular types (*types_unit* is necessary because the formal syntax only permits identifiers on the right side of **where** equations, and not general type expressions). Because *o_unit* can be independently compiled, the code for *o* is not duplicated when the compound unit *c* is created or invoked, and because *o_unit* is a first-class value, it represents *o* as a first-class polymorphic function.

To implement a recursive function, a unit imports the function to be used for recursive calls, and links with itself. Figure 2.5 defines a factorial function with this technique. The unit *fact_unit* defines and exports the function *fact*, which relies on the imported *facti* function for recursive calls. Linking the unit by itself, providing the exported *fact* function for the imported *facti*, creates the function.

The same technique works at the type level to support recursive datatype definitions, such as lists of integers (Figure 2.6). The unit defines type *list* in terms of an opaquely imported type *listi*, and defines the standard constructors and destructors for lists with reference to *list* and *listi* as needed. The compound unit supplies *list* for the *listi* import, causing each of the exported function's type references to *listi* to become references to *list*. Further, it supplies *u*, which equals **int** for the *list_unit*'s type import. Since *list* is exported opaquely, the fact that *list* depends on *listi* is not known at the site of the compound.

Recursive module extensions to ML can typically support one additional example, polymorphic recursion over non-uniform datatypes [Leroy 2003; Dreyer 2005b].[6] ML itself supports non-uniform datatypes, but not polymorphic recursion. Recursive modules add

[6]Polymorphic recursion refers to the recursive call of a polymorphic function at a different type, and a non-uniform datatype is a polymorphic datatype where the specification contains a self-reference at a different type.

```
module o_mod provide o_unit =
  o_unit =
    unit import [t, u, v] export [o:(u→v)→(t→u)→t→v].
      o = λ f. λ g. λ x. f (g x)

module t_mod provide types_unit =
  types_unit =
    unit import [] export [t=int×int, u=int, v=int→int].
      t=int×int
      u=int
      v=int→int

module c_mod provide c_unit =
  c =
    compound import []
             export [t^1=int×int, u^1=int, v^1=int→int,
                     o^0:(u_1→v_1)→(t_1→u_1)→t_1→v_1]
      link o_mod.o_unit : import [t^0, u^0, v^0] export [o^0:...]
           t_mod.types_unit : import [] export [t^1, u^1, v^1]
      where (t^0 ← t^1) (u^0 ← u^1) (v^0 ← v^1)

module math provide add add_tuple =
  add:int→int→int = ...
  add_tuple:int×int→int = ...

module ex provide ex6 =
  invoke c_mod.c as o
  ex6 = (o.o math.add math.add_tuple) (1, 2) 3
```

Figure 2.4. Polymorphic function example: function composition

```
module m provide f4 =
  fact_unit =
    unit import [facti:int→int] export [fact:int→int].
      fact = λ i.
        if0 i then 1 else i × facti (i - 1)

  fact_compound =
    compound import [] export [fact:int→int]
      link fact_unit:import ... export ...
      where facti ← fact

  invoke fact_compound as F
  f4 = F.fact 4
```

Figure 2.5. Recursive function example: factorial

```
module $m$ provide $ex$ $l$ =
  $list\_unit$ =
    unit import [$t$, $listi$] export [$list$, $cons$:$t \to listi \to list$, $nil$:$list$, $hd$:$list \to t$, $tl$:$list \to listi$].
      $list$=int+($t \times listi$)
      $cons$ = $\lambda$ $x$:$t$. $\lambda$ $l$:$listi$. injr ($x$, $l$)
      $nil$:$list$ = injl $0$
      $hd$ = $\lambda$ $l$:$list$. case $l$ of ($\lambda$ $n$. ...) + ($\lambda$ $c$. $\pi_1$ $c$)
      $tl$ = $\lambda$ $l$:$list$. case $l$ of ($\lambda$ $n$. ...) + ($\lambda$ $c$. $\pi_2$ $c$)

  $int\_u$ = unit import [] export [$u$=int]. $u$=int

  $list\_compound$ =
    compound import [] export [$list$, $cons$:int$\to list \to list$, ...]
      link $list\_u$:import ... export ...
           $int\_unit$:import ... export ...
      where ($listi \leftarrow list$) ($t \leftarrow u$)
  invoke $list\_unit$ as $l$
  $ex$ = $l$.$hd$ ($l$.$cons$ $1$ $l$.$nil$)
```

Figure 2.6. Recursive datatype example: list

polymorphic recursion by dispatching the recursive call through the module's signature, which gives the function a fully polymorphic type. In contrast, the type of the direct recursive call is already instantiated to a particular type. The encodings above could be extended to support polymorphic recursion using the same technique, if a unit could export the defined function polymorphically. However, because the formalism does not directly support polymorphic functions or type constructors, these features must be encoded using opaque unit imports.

Figure 2.7 sketches a recursively defined unit that encodes a non-uniform, recursive, polymorphic type constructor *seq* alongside a polymorphic recursive function *size*. (The recursive definition could be encoded as in the above examples with a unit that imports and exports the unit *v*.) Although type correct, *v* diverges when invoked; invocation leads to a recursive invocation of the same unit, although at a different type. The uniform list datatype example did not rely upon self-invocation because the *list* and *listi* types could be directly linked. The need here to keep *seq* and *size* polymorphic requires that they stay in a unit with a type parameter, which must be invoked to render *size* and *seq* accessible. If the example only needed to define *size*, the invocation could be delayed until the recursive call was needed, under a λ-abstraction, which would remedy the divergence problem. This is, however, not an option when defining *seq* as well.

```
v =
  unit import [$a^0$] export [$seq^0$, $size^0$:$seq^0$→int] .
    c =
      compound import [] export [$seq^1$, $size^1$:$seq^1$→int]
        link v : import [$a^1$] export [$seq^1$, $size^1$:$seq^1$→int]
        where ($a^1 \leftarrow a^0 \times a^0$)
    invoke c as m
    $seq^0$ = int+($a^0$×m.seq)
    $size^0$ = $\lambda$ s. case s of ($\lambda$ n. 0) + ($\lambda$ s. $1 + 2{*}m.size(\pi_2\ s)$))
```

Figure 2.7. Polymorphic recursion example: sequences [Okasaki 1998] (diverges)

2.3 Structures and Functors

Units and modules together have a close counterpart in Standard ML's system of structures and functors. However, there are significant differences in how the two systems express similar functionality. In particular, sealing in ML is tied exclusively to the structure construct, whereas it is tied directly to units in my system. Functors, which consume and produce structures, create abstract types by using sealing on those structures. Units, however, support sealing directly because their imports and exports need not be wrapped in modules, and because using sealing to enforce abstraction is a fundamental part of component-oriented programming. Although functor and units both support external linking, there is a significant difference in how. Functors link with application which not only uses the argument structure to satisfy the functor's imports, but it also immediately produces the result structure. The unit system's separation between linking and invocation (i.e., unit linking produces another externally linkable unit, and unit invocation produces an internally linkable module) is a crucial difference between units and functors.

To make the observations of the previous paragraph more precise, Figure 2.8, Table 2.1, and Figure 2.9 define a language based on the same core as the language of Figure 2.1, but with structures and functors rather than modules and units. Section 2.3.2 then shows how to translate programs in the structure and functor language into module and unit programs. The type system for this language is based on Leroy's [1994] formalism of manifest types and on the type system for the language of modules and units (see Chapter 3). The translation needs to know the types of certain variables, so the structure and functor type system also inserts type annotations into the source structure and functor program.

Names, identifiers, and paths:

m	$=$	structure and functor names	
x, t	$=$	value and type names	
P'_m	$=$	$m^i \mid P_m.m$	*paths to structures and functors*
P'_x	$=$	$x^i \mid P_m.x$	*paths to values*
P'_t	$=$	$t^i \mid P_m.t$	*paths to types*
P'	$=$	$P'_m \mid P'_t \mid P'_x$	

Types:

Ty'	$=$	$\mathbf{int} \mid Ty' + Ty' \mid Ty' \times Ty' \mid Ty' \rightarrow Ty'$	*base, sum, product, function types*
	$\mid$	P'_t	*type references*
B'	$=$	$x^i\text{:}Ty'$	*value bindings*
	$\mid$	$t^i \mid t^i\text{=}Ty'$	*opaque and translucent type bindings*
	$\mid$	$m^i\text{:}\mathcal{M}' \mid m^i\text{:}F'$	*structure and functor bindings*
$\mathcal{M}'$	$=$	$\epsilon \mid B',\mathcal{M}'$	*structure types*
F'	$=$	$m^i\text{:}\mathcal{M}' \rightarrow \mathcal{M}'$	*functor types*
Γ'	$=$	$\epsilon \mid \Gamma',B'$	*typing contexts*

Terms:

D'	$=$	$x^i\text{:}Ty'\text{=}E' \mid t^i\text{=}Ty'$	*value and type definitions*
	$\mid$	$\mathbf{structure}\ m^i\text{=}S'$	*structure definition*
	$\mid$	$\mathbf{structure}\ m^i\text{:>}\mathcal{M}'\text{=}S'$	*structure definition with annotation*
	$\mid$	$\mathbf{functor}\ m^i\text{:}F'\text{=}Ds'$	*functor definition*
S'	$=$	$Ds'\text{:>}\mathcal{M}'$	*structure body with sealing*
	$\mid$	$P'_m(P'_m)$	*functor application*
	$\mid$	$P'_m\text{:}F'(P'_m\text{:}\mathcal{M}')\text{:>}\mathcal{M}'$	*functor application with annotations*
Ds'	$=$	$\epsilon \mid D'\ Ds'$	*definition sequences*
E'	$=$	$\mathbb{Z}$	*integer constants*
	$\mid$	P'_x	*value references*
	$\mid$	$\mathbf{injl}\ E' \mid \mathbf{injr}\ E' \mid \mathbf{case}\ E'\ \mathbf{of}\ E' + E'$	*sum expressions*
	$\mid$	$(E',E') \mid \pi_1 E' \mid \pi_2 E'$	*product expressions*
	$\mid$	$\lambda x^i\text{:}Ty'.E' \mid E'E'$	*function expressions*
	$\mid$	$\mathbf{let}\ Ds'\ \mathbf{in}\ E'$	*let expressions*

Figure 2.8. Syntax of structures and functors

Table 2.1. Glossary for the structure and functor type system

Relation	Meaning
$\textsc{distinct}\ \mathcal{M}'$	no names are multiply defined in a structure type
$\Gamma' \vdash P' \mapsto B'$	lookup of a path in a typing context
$\Gamma' \vdash \mathit{Ty}'$	well-formed types
$\Gamma' \vdash B'$	well-formed bindings
$\Gamma' \vdash \mathcal{M}'$	well-formed structure types
$\Gamma' \vdash F'$	well-formed functor types
$\Gamma' \vdash B' <: B'$	subtyping for bindings
$\Gamma' \vdash \mathcal{M}' <: \mathcal{M}'$	subtyping for structure types
$\Gamma' \vdash F' <: F'$	subtyping for functor types
$\Gamma' \vdash \mathit{Ty}' \equiv \mathit{Ty}'$	type equality
$\Gamma' \vdash \mathit{Ty}'\ \delta^{\mathrm{P}}\ \mathit{Ty}'$	type equality that only expands a top-level path
$\Gamma' \vdash D' : B' \leadsto D'$	typing for definitions, adds annotations
$\Gamma' \vdash S' : \mathcal{M}' \leadsto S'$	typing for structures, adds annotations
$\Gamma' \vdash \mathit{Ds}' : \mathcal{M}' \leadsto \mathit{Ds}'$	typing for sequences of definitions, adds annotation
$\Gamma' \vdash E' : \mathit{Ty}' \leadsto E'$	typing for expressions, adds annotations
$\Gamma' \vdash_{\mathrm{P}} E' : \mathit{Ty}' \leadsto E'$	as above, expands types that are paths
Function	**Meaning**
$\sigma : B' \rightarrow \mathit{id}$	gets the identifier of a binding
$\sigma\mathrm{N} : B' \rightarrow n$	gets the name of a binding
$\textsc{dom} : \Gamma' \rightarrow \mathit{ids}$	gets all of the identifiers bound by a typing context
$\textsc{dom} : \mathcal{M}' \rightarrow \mathit{ids}$	gets all of the identifiers bound by a structure type
$\textsc{dom}\mathrm{N} : \mathcal{M}' \rightarrow \mathit{ns}$	gets all of the names bound by a structure type
$\cdot\{\cdot \leftarrow \cdot\} : \mathcal{M}' \times \mathit{id} \times P' \rightarrow \mathcal{M}'$	substitutes a path for an ident. in a structure type

$$\boxed{\Gamma' \vdash P' \mapsto B'}$$

$$\frac{\sigma(B) = id}{\Gamma_1, B, \Gamma_2 \vdash id \mapsto B}\ (\textsc{look}1') \qquad \frac{\Gamma \vdash P \mapsto m^i{:}(\mathcal{M}_1, B, \mathcal{M}_2) \qquad \sigma\textsc{n}(B) = n_1}{\Gamma \vdash P.n_1 \mapsto B\{n_2^j \leftarrow P.n_2 | n_2^j \in \textsc{dom}(\mathcal{M}_1)\}}\ (\textsc{look}2')$$

$$\boxed{\Gamma' \vdash Ty'}$$

$$\frac{}{\Gamma \vdash \mathbf{int}}\ (\textsc{Tint}') \qquad \frac{\Gamma \vdash Ty_1 \qquad \Gamma \vdash Ty_2}{\Gamma \vdash Ty_1 \rightarrow Ty_2}\ (\textsc{Tfun}') \qquad \frac{\Gamma \vdash Ty_1 \qquad \Gamma \vdash Ty_2}{\Gamma \vdash Ty_1 + Ty_2}\ (\textsc{Tsum}')$$

$$\frac{\Gamma \vdash Ty_1 \qquad \Gamma \vdash Ty_2}{\Gamma \vdash Ty_1 \times Ty_2}\ (\textsc{Tprod}') \qquad \frac{(\Gamma \vdash P \mapsto t^i) \vee (\Gamma \vdash P \mapsto t^i{=}Ty)}{\Gamma \vdash P}\ (\textsc{Tpath}')$$

$$\boxed{\Gamma' \vdash B'}$$

$$\frac{\Gamma \vdash Ty}{\Gamma \vdash x^i{:}Ty}\ (\textsc{Bval}') \qquad \frac{}{\Gamma \vdash t^i}\ (\textsc{Btype}1') \qquad \frac{\Gamma \vdash Ty}{\Gamma \vdash t^i{=}Ty}\ (\textsc{Btype}2')$$

$$\frac{\Gamma \vdash \mathcal{M} \qquad \textsc{distinct}\ \mathcal{M}}{\Gamma \vdash m^i{:}\mathcal{M}}\ (\textsc{Bstr}') \qquad \frac{\Gamma \vdash F}{\Gamma \vdash m^i{:}F}\ (\textsc{Bftor}')$$

$$\boxed{\Gamma' \vdash \mathcal{M}'}$$

$$\frac{}{\Gamma \vdash \epsilon}\ (\mathcal{M}\epsilon') \qquad \frac{\Gamma \vdash B \qquad \Gamma, B \vdash \mathcal{M} \qquad \sigma(B) \notin \textsc{dom}(\Gamma)}{\Gamma \vdash B, \mathcal{M}}\ (\mathcal{M}')$$

$$\boxed{\Gamma' \vdash F'}$$

$$\frac{\begin{array}{c} m^i \notin \textsc{dom}(\Gamma) \qquad m \notin \textsc{dom}\mathrm{N}(\mathcal{M}_2) \\ \Gamma \vdash \mathcal{M}_1 \qquad \textsc{distinct}\ \mathcal{M}_1 \qquad \Gamma, (m^i{:}\mathcal{M}_1) \vdash \mathcal{M}_2 \qquad \textsc{distinct}\ \mathcal{M}_2 \end{array}}{\Gamma \vdash m^i{:}\mathcal{M}_1 \rightarrow \mathcal{M}_2}\ (\mathrm{F}')$$

Figure 2.9. Type system for structures and functors

$$\boxed{\Gamma' \vdash B' <: B'}$$

$$\frac{\Gamma \vdash Ty_1 \equiv Ty_2}{\Gamma \vdash x^i{:}Ty_1 <: x^i{:}Ty_2}\ (\textsc{subB1}') \qquad \frac{}{\Gamma \vdash t^i <: t^i}\ (\textsc{subB2}')$$

$$\frac{}{\Gamma \vdash t^i{=}Ty <: t^i}\ (\textsc{subB3}') \qquad \frac{\Gamma \vdash Ty_1 \equiv Ty_2}{\Gamma \vdash t^i{=}Ty_1 <: t^i{=}Ty_2}\ (\textsc{subB4}')$$

$$\frac{\Gamma \vdash \mathcal{M}_1 <: \mathcal{M}_2}{\Gamma \vdash m^i{:}\mathcal{M}_1 <: m^i{:}\mathcal{M}_2}\ (\textsc{subB5}') \qquad \frac{\Gamma \vdash F_1 <: F_2}{\Gamma \vdash m^i{:}F_1 <: m^i{:}F_2}\ (\textsc{subB6}')$$

$$\boxed{\Gamma' \vdash \mathcal{M}' <: \mathcal{M}'}$$

$$\frac{}{\Gamma \vdash \epsilon <: \epsilon}\ (\textsc{subStr1}') \qquad \frac{\Gamma{,}B \vdash \mathcal{M}_1 <: \mathcal{M}_2}{\Gamma \vdash B{,}\mathcal{M}_1 <: \mathcal{M}_2}\ (\textsc{subStr2}')$$

$$\frac{\Gamma \vdash B_1 <: B_2 \qquad \Gamma{,}B_1 \vdash \mathcal{M}_1 <: \mathcal{M}_2}{\Gamma \vdash B_1{,}\mathcal{M}_1 <: B_2{,}\mathcal{M}_2}\ (\textsc{subStr3}')$$

$$\boxed{\Gamma' \vdash F' <: F'}$$

$$\frac{\Gamma \vdash \mathcal{M}_3 <: \mathcal{M}_1 \qquad \Gamma{,}(m^i{:}\mathcal{M}_3) \vdash \mathcal{M}_2 <: \mathcal{M}_4}{\Gamma \vdash m^i{:}\mathcal{M}_1 \to \mathcal{M}_2 <: m^i{:}\mathcal{M}_3 \to \mathcal{M}_4}\ (\textsc{subF}')$$

$$\boxed{\Gamma' \vdash Ty' \equiv Ty'}$$

$$\frac{\Gamma \vdash P \mapsto (t^i{=}Ty)}{\Gamma \vdash P \equiv Ty}\ (\textsc{eqpath}') \qquad \frac{}{\Gamma \vdash Ty \equiv Ty}\ (\textsc{eqrefl}') \qquad \frac{\Gamma \vdash Ty_2 \equiv Ty_1}{\Gamma \vdash Ty_1 \equiv Ty_2}\ (\textsc{eqsym}')$$

$$\frac{\Gamma \vdash Ty_1 \equiv Ty_3 \qquad \Gamma \vdash Ty_3 \equiv Ty_2}{\Gamma \vdash Ty_1 \equiv Ty_2}\ (\textsc{eqtrans}')$$

$$\frac{\Gamma \vdash Ty_1 \equiv Ty_3 \qquad \Gamma \vdash Ty_2 \equiv Ty_4}{\Gamma \vdash Ty_1 \to Ty_2 \equiv Ty_3 \to Ty_4}\ (\textsc{eqfun}')$$

$$\frac{\Gamma \vdash Ty_1 \equiv Ty_3 \qquad \Gamma \vdash Ty_2 \equiv Ty_4}{\Gamma \vdash Ty_1 + Ty_2 \equiv Ty_3 + Ty_4}\ (\textsc{eqprod}')$$

$$\frac{\Gamma \vdash Ty_1 \equiv Ty_3 \qquad \Gamma \vdash Ty_2 \equiv Ty_4}{\Gamma \vdash Ty_1 + Ty_2 \equiv Ty_3 + Ty_4}\ (\textsc{eqsum}')$$

Figure 2.9. continued

$$\boxed{\Gamma' \vdash D' : B' \rightsquigarrow D'}$$

$$\frac{\Gamma \vdash Ty_1 \qquad \Gamma \vdash E_1 : Ty_2 \rightsquigarrow E_2 \qquad \Gamma \vdash Ty_2 \equiv Ty_1}{\Gamma \vdash (x^i\textbf{:}Ty_1\textbf{=}E_1) : (x^i\textbf{:}Ty_1) \rightsquigarrow (x^i\textbf{:}Ty_1\textbf{=}E_2)} \ (\textsc{Dval}')$$

$$\frac{\Gamma \vdash Ty}{\Gamma \vdash (t^i\textbf{=}Ty) : (t^i\textbf{=}Ty) \rightsquigarrow (t^i\textbf{=}Ty)} \ (\textsc{Dtype}')$$

$$\frac{\Gamma \vdash S_1 : \mathcal{M} \rightsquigarrow S_2}{\Gamma\vdash(\textbf{structure } m^i\textbf{=}S_1) : (m^i\textbf{:}\mathcal{M}) \rightsquigarrow (\textbf{structure } m^i\textbf{:>}\mathcal{M}\textbf{=}S_2)} \ (\textsc{Dstr}')$$

$$\frac{\begin{array}{c} m_2^i \notin \textsc{dom}(\Gamma) \qquad m_2 \notin \textsc{dom}\mathrm{N}(\mathcal{M}_2) \\ \Gamma \vdash \mathcal{M}_1 \qquad \textsc{distinct } \mathcal{M}_1 \qquad \Gamma\textbf{,}(m_2^i\textbf{:}\mathcal{M}_1) \vdash \mathcal{M}_2 \qquad \textsc{distinct } \mathcal{M}_2 \\ \Gamma\textbf{,}(m_2^i\textbf{:}\mathcal{M}_1) \vdash Ds_1 : \mathcal{M}_3 \rightsquigarrow Ds_2 \qquad \Gamma\textbf{,}(m_2^i\textbf{:}\mathcal{M}_1) \vdash \mathcal{M}_3 <: \mathcal{M}_2 \end{array}}{\begin{array}{c}\Gamma \vdash (\textbf{functor } m_1^i\textbf{:(}m_2^i\textbf{:}\mathcal{M}_1 \to \mathcal{M}_2\textbf{)=}Ds_1) : (m_1^i\textbf{:(}m_2^i\textbf{:}\mathcal{M}_1 \to \mathcal{M}_2\textbf{)}) \rightsquigarrow \\ (\textbf{functor } m_1^i\textbf{:(}m_2^i\textbf{:}\mathcal{M}_1 \to \mathcal{M}_2\textbf{)=}Ds_2)\end{array}} \ (\textsc{Dftor}')$$

$$\boxed{\Gamma' \vdash S' : \mathcal{M}' \rightsquigarrow S'}$$

$$\frac{\Gamma \vdash Ds_1 : \mathcal{M}_2 \rightsquigarrow Ds_2 \qquad \Gamma \vdash \mathcal{M}_1 \qquad \textsc{distinct } \mathcal{M}_2 \qquad \Gamma \vdash \mathcal{M}_2 <: \mathcal{M}_1}{\Gamma \vdash (Ds_1\textbf{:>}\mathcal{M}_1) : \mathcal{M}_1 \rightsquigarrow (Ds_2\textbf{:>}\mathcal{M}_1)} \ (\textsc{Sbody}')$$

$$\frac{\begin{array}{c}\Gamma \vdash P_1 \mapsto m_1^i\textbf{:}F \qquad \Gamma \vdash P_2 \mapsto m_2^i\textbf{:}\mathcal{M}_1 \\ F = m_3^i\textbf{:}\mathcal{M}_2 \to \mathcal{M}_3 \qquad \Gamma \vdash \mathcal{M}_1 <: \mathcal{M}_2 \qquad \mathcal{M}_4 = \mathcal{M}_3\{m_3^i \leftarrow P_2\}\end{array}}{\Gamma \vdash (P_1\textbf{(}P_2\textbf{)}) : \mathcal{M}_4 \rightsquigarrow (P_1\textbf{:}F\textbf{(}P_2\textbf{:}\mathcal{M}_1\textbf{):>}\mathcal{M}_4)} \ (\textsc{Sapp}')$$

$$\boxed{\Gamma' \vdash Ds' : \mathcal{M}' \rightsquigarrow Ds'}$$

$$\frac{}{\Gamma \vdash \epsilon : \epsilon \rightsquigarrow \epsilon} \ (\textsc{Ds}\epsilon')$$

$$\frac{\Gamma \vdash D_1 : B \rightsquigarrow D_2 \qquad \Gamma\textbf{,}B \vdash Ds : \mathcal{M} \rightsquigarrow Ds_2 \qquad \sigma(B) \notin \textsc{dom}(\Gamma)}{\Gamma \vdash (D_1\ Ds_1) : (B\textbf{,}\mathcal{M}) \rightsquigarrow (D_2\ Ds_2)} \ (\textsc{Ds}')$$

Figure 2.9. continued

$$\boxed{\Gamma' \vdash E' : Ty' \rightsquigarrow E'}$$

$$\frac{z \in \mathbb{Z}}{\Gamma \vdash z : \mathbf{int} \rightsquigarrow z}\ (\textsc{Eint}') \qquad \frac{\Gamma \vdash P \mapsto x^i\mathbf{:}Ty}{\Gamma \vdash P : Ty \rightsquigarrow P}\ (\textsc{Epath}')$$

$$\frac{\Gamma \vdash E_1 : Ty_1 \rightsquigarrow E_2 \qquad \Gamma \vdash Ty_2}{\Gamma \vdash \mathbf{injl}\ E_1 : Ty_1 + Ty_2 \rightsquigarrow \mathbf{injl}\ E_2}\ (\textsc{Einjl}')$$

$$\frac{\Gamma \vdash E_1 : Ty_2 \rightsquigarrow E_2 \qquad \Gamma \vdash Ty_1}{\Gamma \vdash \mathbf{injr}\ E_1 : Ty_1 + Ty_2 \rightsquigarrow \mathbf{injr}\ E_2}\ (\textsc{Einjr}')$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{P}} E_1 : Ty_1 + Ty_2 \rightsquigarrow E_4 \\ \Gamma \vdash_{\mathrm{P}} E_2 : Ty_3 \to Ty_4 \rightsquigarrow E_5 \qquad \Gamma \vdash_{\mathrm{P}} E_3 : Ty_5 \to Ty_6 \rightsquigarrow E_6 \\ \Gamma \vdash Ty_1 \equiv Ty_3 \qquad \Gamma \vdash Ty_2 \equiv Ty_5 \qquad \Gamma \vdash Ty_4 \equiv Ty_6\end{array}}{\Gamma \vdash \mathbf{case}\ E_1\ \mathbf{of}\ E_2 + E_3 : Ty_4 \rightsquigarrow \mathbf{case}\ E_4\ \mathbf{of}\ E_5 + E_6}\ (\textsc{Ecase}')$$

$$\frac{\Gamma \vdash E_1 : Ty_1 \rightsquigarrow E_3 \qquad \Gamma \vdash E_2 : Ty_2 \rightsquigarrow E_4}{\Gamma \vdash \mathbf{(}E_1\mathbf{,}\ E_2\mathbf{)} : Ty_1 \times Ty_2 \rightsquigarrow \mathbf{(}E_3\mathbf{,}\ E_4\mathbf{)}}\ (\textsc{Eprod}')$$

$$\frac{\Gamma \vdash_{\mathrm{P}} E_1 : Ty_1 \times Ty_2 \rightsquigarrow E_2}{\Gamma \vdash \pi_1 E_1 : Ty_1 \rightsquigarrow \pi_1 E_2}\ (\textsc{Epj1}') \qquad \frac{\Gamma \vdash_{\mathrm{P}} E_1 : Ty_1 \times Ty_2 \rightsquigarrow E_2}{\Gamma \vdash \pi_2 E_1 : Ty_2 \rightsquigarrow \pi_2 E_2}\ (\textsc{Epj2}')$$

$$\frac{x^i \notin \textsc{dom}(\Gamma) \qquad \Gamma \vdash Ty_1 \qquad \Gamma\mathbf{,}x^i\mathbf{:}Ty_1 \vdash E_1 : Ty_2 \rightsquigarrow E_2}{\Gamma \vdash (\lambda x^i\mathbf{:}Ty_1\mathbf{.}E_1) : (Ty_1 \to Ty_2) \rightsquigarrow (\lambda x^i\mathbf{:}Ty_1\mathbf{.}E_2)}\ (\textsc{Efun}')$$

$$\frac{\Gamma \vdash_{\mathrm{P}} E_1 : Ty_1 \to Ty_2 \rightsquigarrow E_3 \qquad \Gamma \vdash E_2 : Ty_3 \rightsquigarrow E_4 \qquad \Gamma \vdash Ty_1 \equiv Ty_3}{\Gamma \vdash E_1\ E_2 : Ty_2 \rightsquigarrow E_3\ E_4}\ (\textsc{Eapp}')$$

$$\frac{\begin{array}{c}\Gamma \vdash Ds_1 : \mathcal{M} \rightsquigarrow Ds_2 \\ \Gamma\mathbf{,}\mathcal{M} \vdash E_1 : Ty_1 \rightsquigarrow E_2 \qquad \Gamma\mathbf{,}\mathcal{M} \vdash Ty_1 \equiv Ty_2 \qquad \Gamma \vdash Ty_2\end{array}}{\Gamma \vdash (\mathbf{let}\ Ds_1\ \mathbf{in}\ E_1) : Ty_2 \rightsquigarrow (\mathbf{let}\ D_2\ \mathbf{in}\ E_2)}\ (\textsc{Elet}')$$

$$\boxed{\Gamma' \vdash_{\mathrm{P}} E' : Ty' \rightsquigarrow E'}$$

$$\frac{\Gamma \vdash E_1 : Ty_1 \rightsquigarrow E_2 \qquad \Gamma \vdash Ty_1\ \delta^{\mathrm{P}}\, Ty_2 \qquad Ty_2 \text{ is not a path } (P)}{\Gamma \vdash_{\mathrm{P}} E_1 : Ty_2 \rightsquigarrow E_2}$$

Figure 2.9. continued

The structure and functor language has only generative functors, which Standard ML uses, and not applicative functors [Leroy 1995], which OCaml uses. In a generative functor system, the opaque types produced from any two functor applications are different; in other words, each functor application generates new types. Units are similarly generative; two invocations, even of the same unit, create two incompatible types for each opaque type export. In an applicative functor system, every application of a functor to equivalent arguments yields opaque types that are known to be equal. I do not address applicative functors in this work.

The language of Figure 2.8 departs from SML's generative first-order functors in the following ways.

- Structure and functor definitions can be freely nested. Although my structure and functor language does not directly support higher-order functors (i.e., functors as the parameter to and result of another functor), the same effect can be achieved by importing and exporting structures that contain functor definitions. Thus, Section 2.3.2's translation gives a suitable account of higher-order functors.
- Each structure definition must be given an explicit structure type, whereas in SML, most general signatures are inferred for a structure that lacks an explicitly given type.
- There are no `where` or `sharing` clauses for structure types, and the argument expression in a functor application must be a path (as opposed to an arbitrary structure expression). Leroy [1996] shows that these limitations do not impact expressiveness.
- The functor expression in a functor application must be a path.
- Structures cannot be renamed (e.g., `structure x = y` in SML). Support for this feature requires a mechanism to redirect all opaque type definitions in `x` to their origination in `y`; this mechanism is known as *strengthening* or *selfification*.

2.3.1 Diamond Imports

An example of the well-known diamond import pattern in both systems serves to illustrate their correspondence prior to the presentation of the formal translation. *Diamond import* refers to a linking pattern where two functors are instantiated with the same argument, and their results are subsequently linked together. In the final linking step, the initial argument should be known to be the same for both modules being linked. In SML, sharing constraints are used to solve diamond import problems; however, translucent types are also sufficient to solve the problem. The lack of translucent types prevented diamond imports from being expressible in Flatt and Felleisen's unit system.

A typical example of a diamond import is a parser, set up as in Figure 2.10 (using multiple argument functors to simplify the example). The *front_end* functor imports the types t and u; however, $P.u$ is declared equal to $L.t$, allowing *front_end*'s body to rely on this equality, but requiring the functor's arguments to satisfy it. The key to this example is that the type *symbols*.t is written down at a single place, and the rest of the example systematically refers to it using type equations.

The same techniques apply to a corresponding unit-based program (Figure 2.11, some type annotations and identifier stamps are omitted). Without translucent type exports, *symbols_unit* would have to export t opaquely, leading l and p to export t and u opaquely, which would prevent them from linking with *front_end*. Flatt and Felleisen's unit system worked around this deficiency by linking *lexer*, *parser*, *front_end* and *symbols_unit* together all in a single **compound** expression, so that the opaquely exported type t from *symbols_unit* is directly imported into each of the other units (including *front_end*). All of the symbol types t are known to be the same since they arise from the same place. Translucency allows the information about the equality of the opaque types to be kept and propagated, which enables the incremental style of linking of Figure 2.11.

2.3.2 Translating Structures and Functors

Table 2.2 and Figure 2.12 present a translation from the language of structures and functors to the language of modules and units (the translation relations omitted from Figure 2.12 are all straightforward structural traversals). A structure definition translates

```
structure symbols :> [t] = (t=...) ...
functor lexer : (S:[t]→[t=S.t,get_tok:...t...]) =...
functor parser : (S:[u]→[u=S.u,do_parse:...u...]) =...
functor front_end : (L:[t,get_tok:...t...],
                     P:[u=L.t,do_parse:...u...]→...) =...
structure l:>[t=symbols.t,get_tok:...t...]=lexer(symbols)
structure p:>[u=symbols.t,do_parse:...u...]=parser(symbols)
structure ...= front_end(l,p)
```

Figure 2.10. Diamond import example: structures and functors

```
invoke (unit import [] export [t]. t = ...) as symbols:...
symbols_unit = unit import [] export [t=symbols.t]. t=symbols.t
lexer = unit import [St] export [t=St, get_tok:...t...]. ...
parser = unit import [Su] export [u=Su, do_parse:...u...]. ...
front_end = unit import [Lt, get_tok:...Lt..., Pu=Lt, do_parse:...Pu...]
                      export [...]. ...
l = compound import [] export [t^2=symbols.t, get_tok:...t^2...]
    link lexer:import [St] export [t^2=St, get_tok:...t^2...]
         symbols_unit:import [] export [t^1=symbols.t]
    where St←t^1
p = compound import [] export [u=symbols.t, do_parse:...u...]
    link parser:import [Su] export [u=Su, do_parse:...u...]
         symbols_unit:import [] export [t=symbols.t]
    where Su←t
invoke (compound import [] export [...]
         link l:import [] export [t=symbols.t, get_tok^1:...t...]
              p:import [] export [u=symbols.t, do_parse^1:...u...]
              front_end:import [Lt, get_tok^2:...Lt..., Pu=Lt, do_parse^2:...Pu...]
                        export ...
         where (Lt←t) (get_tok^2←get_tok^1)
               (Pu←u) (do_parse^2←do_parse^1)) as ...
```

Figure 2.11. Diamond import example: modules and units

Table 2.2. Glossary for the structure and functor to module and unit translation

Relation	Meaning
$P' \Longrightarrow P$	translates s.f. paths to m.u. paths
$Ty' \Longrightarrow Ty$	translates types
$B' \Longrightarrow \mathcal{M}$	translates s.f. bindings to sequences of m.u. bindings
$\mathcal{M}' \Longrightarrow \mathcal{M}$	translates structure types to module descriptions
$F' \Longrightarrow Ty$	translates functor types to of m.u. types
$\Gamma' \Longrightarrow \Gamma$	translates typing contexts
$D' \Longrightarrow Ds$	translates s.f. definitions to sequences of m.u. def'ns
$S' \Longrightarrow E$	translates structures to unit expressions
$Ds' \Longrightarrow Ds$	translates definition sequences
$E' \Longrightarrow E$	translates expressions
Function	**Meaning**
$\textsc{domN} : \mathcal{M} \to n \ldots n$	gets all of the names bound by a module description
$\textsc{strengthen} : P_m \times \mathcal{M} \to \mathcal{M}$	redirects any opaque types to the path
$\textsc{strengthen} : P_m \times B \to B$	redirects any opaque types to the path
$\textsc{structUnit} : m^i \times \mathcal{M} \to D$	builds a unit that exports the module description, uses the types in m^i
$\mathcal{M}\textsc{toDs} : P_m \times \mathcal{M} \to Ds$	builds definitions that have the given module descr., use the definitions in the path
$\textsc{BtoD} : P_m \times B \to D$	builds a definition that has the given binding, uses the definitions in the path

$$\boxed{B' \Longrightarrow \mathcal{M}}$$

$$\frac{Ty' \Longrightarrow Ty}{x^i{:}Ty' \Longrightarrow [x^i{:}Ty]} \qquad \frac{}{t^i \Longrightarrow [t^i]} \qquad \frac{Ty' \Longrightarrow Ty}{t^i{=}Ty' \Longrightarrow [t^i{=}Ty]}$$

$$\frac{\begin{array}{c} \mathcal{M}' \Longrightarrow \mathcal{M}_1 \qquad B_1 = m^i{:}\mathcal{M}_1\ (\textsc{domN}(\mathcal{M}_1)) \\ \mathcal{M}_2 = \textsc{strengthen}(m^i, \mathcal{M}_1) \qquad B_2 = \mathit{exp\_mod}^j{:}\mathcal{M}_2\ (\textsc{domN}(\mathcal{M}_2)) \end{array}}{m^i{:}\mathcal{M}' \Longrightarrow [B_1,(\mathit{m\_unit}^i{:}\mathbf{unitT}\ [B_2]\ (\epsilon \rightarrow \mathit{exp\_mod}))]}$$

$$\frac{F' \Longrightarrow Ty}{m^i{:}F' \Longrightarrow \mathit{m\_unit}^i{:}Ty}$$

$$\boxed{F' \Longrightarrow Ty}$$

$$\frac{\mathcal{M}_1' \Longrightarrow \mathcal{M}_1 \qquad \mathcal{M}_2' \Longrightarrow \mathcal{M}_2 \qquad ns = \textsc{domN}(\mathcal{M}_2) \qquad B = m^i{:}\mathcal{M}_1\ (\textsc{domN}(\mathcal{M}_1))}{m^i{:}\mathcal{M}_1' \rightarrow \mathcal{M}_2' \Longrightarrow \mathbf{unitT}\ (B,\mathcal{M}_2)\ (m \rightarrow ns)}$$

$$\boxed{D' \Longrightarrow Ds}$$

$$\frac{E' \Longrightarrow E \qquad Ty' \Longrightarrow Ty}{x^i{:}Ty'{=}E' \Longrightarrow x^i{:}Ty{=}E}\ (\textsc{trans}\mathrm{D1}) \qquad \frac{Ty' \Longrightarrow Ty}{t^i{=}Ty' \Longrightarrow t^i{=}Ty}\ (\textsc{trans}\mathrm{D2})$$

$$\frac{\mathcal{M}' \Longrightarrow \mathcal{M} \qquad S' \Longrightarrow E \qquad D = \textsc{structUnit}(m^i, \mathcal{M})}{\mathbf{structure}\ m^i{:>}\mathcal{M}'{=}S' \Longrightarrow (\mathbf{invoke}\ E\ \mathbf{as}\ m^i{:}(\mathcal{M}))\ D}\ (\textsc{trans}\mathrm{D3})$$

$$\frac{\begin{array}{c} F' = (m_2^j{:}\mathcal{M}_1' \rightarrow \mathcal{M}_2') \qquad F' \Longrightarrow Ty \qquad \mathcal{M}_1' \Longrightarrow \mathcal{M}_1 \\ \mathcal{M}_2' \Longrightarrow \mathcal{M}_2 \qquad Ds' \Longrightarrow Ds \qquad B = m_2^j{:}\mathcal{M}_1\ (\textsc{domN}(\mathcal{M}_1)) \end{array}}{\mathbf{functor}\ m_1^i{:}F'{=}Ds' \Longrightarrow \mathit{m_1\_unit}^i{:}Ty{=}\mathbf{unit\ import}\ B\ \mathbf{export}\ \mathcal{M}_2\mathbf{.}\ Ds}\ (\textsc{trans}\mathrm{D4})$$

$$\boxed{S' \Longrightarrow E}$$

$$\frac{Ds' \Longrightarrow Ds \qquad \mathcal{M}' \Longrightarrow \mathcal{M} \qquad E = \mathbf{unit\ import}\ \epsilon\ \mathbf{export}\ \mathcal{M}\mathbf{.}\ Ds}{Ds'{:>}\mathcal{M}' \Longrightarrow E}\ (\textsc{trans}\mathrm{S1})$$

$$\frac{\begin{array}{c} F' = (m_1^i{:}\mathcal{M}_3' \rightarrow \mathcal{M}_4') \\ \mathcal{M}_1' \Longrightarrow \mathcal{M}_1 \qquad \mathcal{M}_2' \Longrightarrow \mathcal{M}_2 \qquad \mathcal{M}_3' \Longrightarrow \mathcal{M}_3 \qquad \mathcal{M}_4' \Longrightarrow \mathcal{M}_4 \\ B_1 = m^i{:}\mathcal{M}_3\ (\textsc{domN}(\mathcal{M}_3)) \qquad IE_1 = \mathbf{import}\ [B_1]\ \mathbf{export}\ \mathcal{M}_4 \\ B_2 = \mathit{exp\_mod}^j{:}\mathcal{M}_1\ (\textsc{domN}(\mathcal{M}_1)) \qquad IE_2 = \mathbf{import}\ \epsilon\ \mathbf{export}\ [B_2] \\ E = \mathbf{compound\ import}\ \epsilon\ \mathbf{export}\ \mathcal{M}_2\ \mathbf{link}\ \mathit{P_2\_unit}{:}IE_2\ \mathit{P_1\_unit}{:}IE_1 \\ \mathbf{where}\ m^i \leftarrow \mathit{exp\_mod}^j \end{array}}{P_1{:}F'(P_2{:}\mathcal{M}_1'){:>}\mathcal{M}_2' \Longrightarrow E}\ (\textsc{trans}\mathrm{S2})$$

Figure 2.12. Translation from structures and functors to modules and units

$$\boxed{\mathcal{M}\text{TODS} : P_m \times \mathcal{M} \to Ds}$$

$$\begin{aligned}
\mathcal{M}\text{TODS}(P, \epsilon) &= \epsilon \\
\mathcal{M}\text{TODS}(P, (B\mathbf{,}\mathcal{M})) &= \text{BTOD}(P, B)\ \mathcal{M}\text{TODS}(P, \mathcal{M})
\end{aligned}$$

$$\boxed{\text{BTOD} : P_m \times B \to D}$$

$$\begin{aligned}
\text{BTODS}(P, x^i\mathbf{:}Ty) &= x^i\mathbf{:}Ty\mathbf{=}P\mathbf{.}x \\
\text{BTODS}(P, t^i) &= t^i\mathbf{=}P\mathbf{.}t \\
\text{BTODS}(P, t^i\mathbf{=}Ty) &= t^i\mathbf{=}Ty \\
\text{BTODS}(P, m^i\mathbf{:}\mathcal{M}\ \mathbf{(}ns\mathbf{)}) &= \mathbf{module}\ m^i\ \mathbf{provide}\ ns\ \mathbf{=}\ \mathcal{M}\text{TODS}(P\mathbf{.}\widehat{m}, \mathcal{M})
\end{aligned}$$

$$\boxed{\text{STRENGTHEN} : P_m \times \mathcal{M} \to \mathcal{M}}$$

$$\begin{aligned}
\text{STRENGTHEN}(P, \epsilon) &= \epsilon \\
\text{STRENGTHEN}(P, (B\mathbf{,}\mathcal{M})) &= \text{STRENGTHEN}(P, B)\mathbf{,}\text{STRENGTHEN}(P, \mathcal{M})
\end{aligned}$$

$$\boxed{\text{STRENGTHEN} : P_m \times B \to B}$$

$$\begin{aligned}
\text{STRENGTHEN}(P, x^i\mathbf{:}Ty) &= x^i\mathbf{:}Ty \\
\text{STRENGTHEN}(P, t^i) &= t^i\mathbf{=}P\mathbf{.}t \\
\text{STRENGTHEN}(P, t^i\mathbf{=}Ty) &= t^i\mathbf{=}Ty \\
\text{STRENGTHEN}(P, m^i\mathbf{:}\mathcal{M}\ \mathbf{(}ns\mathbf{)}) &= m^i\mathbf{:}\text{STRENGTHEN}(P\mathbf{.}\widehat{m}, \mathcal{M})\ \mathbf{(}ns\mathbf{)}
\end{aligned}$$

$$\boxed{\text{STRUCTUNIT} : m^i \times \mathcal{M} \to D}$$

$$\frac{\begin{array}{c} Ds = \mathcal{M}\text{TODS}(m^i, \mathcal{M}_1) \\ \mathcal{M}_2 = \text{STRENGTHEN}(m^i, \mathcal{M}_1) \qquad B = exp\_mod^j\mathbf{:}\mathcal{M}_2\ \mathbf{(}\text{DOMN}(\mathcal{M}_2)\mathbf{)} \\ M = \mathbf{module}\ exp\_mod^j\ \mathbf{provide}\ \text{DOMN}(\mathcal{M}_1)\ \mathbf{=}\ Ds \qquad Ty = \mathbf{unitT}\ B\ \mathbf{(}\to exp\_mod\mathbf{)} \end{array}}{\text{STRUCTUNIT}(m^i, \mathcal{M}_1) = m\_unit^i\mathbf{:}Ty\mathbf{=unit\ import}\ \epsilon\ \mathbf{export}\ B\mathbf{.}\ M}$$

Figure 2.12. continued

into two definitions, a module definition and a unit definition. The module is created by invoking a unit whose exports are the same as the structure's. By going through a unit, the structure's body is appropriately sealed. However, this unit cannot be used as the argument in a functor application; the semantics of a structure definition require that the structure's body is evaluated once, and that opaque type exports are generated once. Using the unit more than once will create multiple types and run the body multiple times.

To ensure single instantiation of a structure-representing unit, a separate unit is built specifically for use in functor applications (the *_unit* suffix is appended to the name to distinguish between the module and the unit). This unit has the same imports and exports as the invoked unit, but each definition in its body simply redirects to the definition provided by the invoked unit. Furthermore, the functor-argument unit replaces all of the opaque exports in its signature with translucent ones that refer to the invoked unit. This operation is similar to the "strengthening" or "selfification" used to support structure renaming. The STRUCTUNIT function builds the functor-argument unit using the ΓTODS function to construct the body from the old export context, and using the STRENGTHEN function to construct the new export context from the old one.

The following example demonstrates the translation from a structure to a module and unit.

```
structure m :> [t, x:t] =
  (t=int
  x:t=1):>[t, x:t]
⟹
invoke (unit import [] export [t, x:t].
          t=int
          x:t=1)
  as m:([t, x:t])
m_unit:...=
  unit import [] export [(exp_mod:[t=m.t, x:t] (x t))].
    module exp_mod provide x t =
      t=m.t
      x:t=m.x
```

A functor definition translates into the definition of a unit whose imports and exports correspond to the functor's imports and exports. Because a functor imports a single structure that groups all of its imported values and types, the unit imports a single module that performs the same grouping. A functor application becomes a compound unit that links the argument structure's unit to the functor's import. The structure's unit exports its bindings under the single module named *exp_mod* to facilitate functor application and because functors produce structures as well as consuming them.

```
functor f:(i:[t, x:t]→[t=i.t, y:t]) =
  t=i.t
  y:t=x
structure s :> [t=m.t, y:t] = f:...(m:...)
⟹
f:... =
  unit import [(i:[t, x:t] (x t))] export [t=i.t, y:t].
    t=i.t
    y:t=x
invoke (compound import [] export [y:t, t=m.t]
          link f:import ... export ...
               m_unit:import ... export ...
          where i←exp_mod) as s:(...)
s_unit:...= ...
```

The functor application construct requires that the paths to the functor and structure are annotated with their types. These annotations are used to create the unit types in the **link** section of the compound unit, and the functor's parameter is further used in the **link** clause, since unit linking is by name.

The transformation can produce programs whose size is quadratic in the size of the input. This is due to the STRENGTHEN function. Consider a module description that contains a nesting of modules n levels deep where the innermost module contains n opaque type specifications. The strengthened version contains n translucent type definitions, each of which contains a path of length n. The same considerations apply to the $\mathcal{M}$TODS function used in STRUCTUNIT, but for value as well as type definitions. For programs with a constant bound on module nesting depth, the translation is linear because each unit constructed by STRUCTUNIT has the same number of definitions as the original structure, and the size of each definition is bounded by the module nesting depth; the other component of the translation's results are essentially rearrangements of the input.

Theorem 2.1 states that if a structure and functor program has a type, then the result of the translation $\Longrightarrow$ has a type that is related to the original type by the translation.[7] I assume that source programs do not contain identifiers ending in *_unit* to avoid collisions between names in the original program and the generated names that end in *_unit*.

[7]These lemmas, theorems and proof sketches rely on the formal semantics of the module and unit language developed in Chapter 3.

Lemma 2.1 (Strengthen)**.** If Γ is well-formed (there exists $\mathcal{M}$ such that $\epsilon \vdash \mathcal{M} : \Gamma$) then

1. $\Gamma \vdash P \mapsto m^i{:}\mathcal{M}\ (ns)$ implies $\Gamma \vdash \textsc{strengthen}(P, \mathcal{M})$
2. $\Gamma \vdash P \mapsto B$ implies $\Gamma \vdash \textsc{strengthen}(P, B)$
3. $\Gamma; \epsilon \vdash P \mapsto m^i{:}\mathcal{M}\ (ns)$ implies $\Gamma \vdash \mathcal{M}\textsc{toDs}(m^i, \mathcal{M}) : \textsc{strengthen}(P, \mathcal{M})$
4. $\Gamma; \epsilon \vdash P \mapsto B$ implies $\Gamma \vdash \textsc{BtoDs}(m^i, B) : \textsc{strengthen}(P, B)$.

Proof. By structural induction on B and $\mathcal{M}$. □

Lemma 2.2 (Context translation)**.** If Γ' is well-formed and $\Gamma' \Longrightarrow \Gamma$ then

1. $\Gamma' \vdash Ty'$ and $Ty' \Longrightarrow Ty$ implies $\Gamma \vdash Ty$
2. $\Gamma' \vdash \mathcal{M}'$ and $\mathcal{M}' \Longrightarrow \mathcal{M}$ implies $\Gamma \vdash \mathcal{M}$
3. $\Gamma' \vdash F'$ and $F' \Longrightarrow Ty$ implies $\Gamma \vdash Ty$
4. $\Gamma' \vdash B'$ and $B' \Longrightarrow \mathcal{M}$ implies $\Gamma \vdash \mathcal{M}$.

Proof. By induction on the translation relations, using Lemma 2.1. □

Lemma 2.3 (Translation subtyping)**.** If Γ' is well-formed and $\Gamma' \Longrightarrow \Gamma$ then

1. $\Gamma' \vdash Ty'_1 \equiv Ty'_2$ and $Ty'_1 \Longrightarrow Ty_1$ and $Ty'_2 \Longrightarrow Ty_2$ implies $\Gamma_1 \vdash Ty_1 <: Ty_2$
2. $\Gamma' \vdash \mathcal{M}'_1 <: \mathcal{M}'_2$ and $\mathcal{M}'_1 \Longrightarrow \mathcal{M}_1$ and $\mathcal{M}'_2 \Longrightarrow \mathcal{M}_2$ implies $\Gamma \vdash \mathcal{M}_1 <: \mathcal{M}_2$
3. $\Gamma' \vdash F'_1 <: F'_2$ and $F'_1 \Longrightarrow Ty_1$ and $F'_2 \Longrightarrow Ty_2$ implies $\Gamma_1 \vdash Ty_1 <: Ty_2$
4. $\Gamma' \vdash B'_1 <: B'_2$ and $B'_1 \Longrightarrow \mathcal{M}_1$ and $B'_2 \Longrightarrow \mathcal{M}_2$ implies $\Gamma_1 \vdash \mathcal{M}_1 <: \mathcal{M}_2$.

Lemma 2.4 (Translation lookup)**.** If Γ' is well-formed and $\Gamma' \Longrightarrow \Gamma$ then

1. $\Gamma' \vdash P \mapsto m^i{:}\mathcal{M}'$ and $\mathcal{M}' \Longrightarrow \mathcal{M}$ and
   $\mathcal{M}_2 = [\mathit{exp\_mod}^j{:}(\textsc{strengthen}(m^i, \mathcal{M})\ (\textsc{dom}\mathrm{N}(\mathcal{M})))]$ implies
   $\Gamma \vdash P\_\mathrm{unit} \mapsto m\_\mathrm{unit}^i{:}\mathbf{unitT}\ \mathcal{M}_2\ (\epsilon \to \mathit{exp\_mod})$
2. $\Gamma' \vdash P \mapsto m^i{:}F'$ and $F' \Longrightarrow Ty$ and implies $\Gamma \vdash P\_\mathrm{unit} \mapsto m\_\mathrm{unit}^i{:}Ty$.

Proof sketch. By structural induction on P, with a structural induction on Γ' in the base case (when P is an identifier). □

Theorem 2.1. If Γ' is well-formed (for some $\mathcal{M}'$, $\epsilon \vdash \mathcal{M}' : \Gamma'$) and $\Gamma' \Longrightarrow \Gamma$ then

1. $D'_1 \Longrightarrow Ds$ and $\Gamma' \vdash D'_2 : B' \leadsto D'_1$ and $B' \Longrightarrow \mathcal{M}$ implies $\Gamma; \epsilon \vdash Ds : \mathcal{M}$
2. $S'_1 \Longrightarrow E$ and $\Gamma' \vdash S'_2 : \mathcal{M}' \leadsto S'_1$ and $\mathcal{M}' \Longrightarrow \mathcal{M}$ implies
   $\Gamma \vdash E : \mathbf{unitT}\ \mathcal{M}\ (\epsilon \to \textsc{dom}\mathrm{N}(\mathcal{M}))$
3. $Ds'_1 \Longrightarrow Ds_1$ and $\Gamma' \vdash Ds'_2 : \mathcal{M}' \leadsto Ds'_1$ and $\mathcal{M}' \Longrightarrow \mathcal{M}$ implies $\Gamma; \epsilon \vdash Ds_1 : \mathcal{M}$
4. $E'_1 \Longrightarrow E_1$ and $\Gamma' \vdash E'_2 : Ty' \leadsto E'_1$ and $Ty' \Longrightarrow Ty$ implies $\Gamma \vdash E_1 : Ty$.

Proof sketch. The proof proceeds by rule induction on the translation relation for D'_1, S'_1, Ds'_1, and E'_1.

case ($\textsc{trans}$D1): By lemmas 2.2 and 2.3.

case ($\textsc{trans}$D2): By lemma 2.2.

case (TRANSD3): By lemma 2.1.

case (TRANSD4): By lemma 2.3.

case (TRANSS1): By lemmas 2.2 and 2.3.

case (TRANSS2): By lemmas 2.2 and 2.4.

□

2.3.3 Cyclic Linking Dependencies

Although SML's generative functors did not originally support cyclic linking dependencies (recursive linking), there are several proposals for and implementations of recursive functor extensions [Crary et al. 1999; Russo 2001; Dreyer 2005b]. Units avoid three significant problems faced by these extensions.

2.3.3.1 Avoiding double vision. In his thesis, Dreyer describes a potentially serious problem for recursive functors that he dubs the *double vision* problem [Dreyer 2005b]; he also notices that units avoid the problem. When a functor imports a function whose type contains an abstract type that is defined by the functor itself, the type system needs to match up the defined type with the import's type. The inability to do so, i.e., the double-vision problem, is illustrated by the following simple example (which does not typecheck).

```
dv_unit=
  unit import [t=int, x:t] export [u, y:u, f:int→u]
    u=int
    y:u=1
    f:int→u=λ z:int. x+z
dv_cmpd=
  compound import [] export [u, f:int→u]
    link dv_unit:import [t=int, x:t] export [u, y:u, f:int→u]
    where (t←u) (x←y)
```

The intent is that the exported type u should be opaque to the outside, and that the imported type t should be known as an **int** inside of the module. However, at the **compound**, u is opaque, and not known to be an **int**. Changing t=**int** into an opaque export, t, would allow the compound unit to check, but results in an ill-typed body of u. Using t=u as the import instead of t=**int** makes both the unit and compound unit well-typed. An import referring to an export this way is not supported by functors.

2.3.3.2 Bootstrapped heap. Okasaki [1998] describes a *bootstrapped heap* data structure with excellent worst-case execution times.[8] The key property of bootstrapped heaps is that the type of an element of a heap is defined in terms of the type of heaps themselves as follows:

```
datatype 'a Heap = E | H of 'a * ('a Heap) PrimH.T
```

where `PrimH` is a structure that defines a primitive, nonbootstrapped heap. The definitions of functions on the `Heap` type have corresponding references to functions from the `PrimH` structure.

Okasaki shows how to parameterize the bootstrapped heap over the primitive heap using a recursive structure definition and a functor `MakeH` that makes primitive heaps (a functor that makes a primitive heap, given an element type, is used because the bootstrapped heap needs to instantiate the type of elements in the primitive heap).

```
structure rec BootstrappedElem =
  struct
    datatype T = E | H of Elem.T * PrimH.Heap
    fun compare = ... Elem.compare ...
  end
and PrimH = MakeH(BootstrappedElem)
```

This idiom is difficult to support with recursive generative functors (although recursive applicative functors can support them naturally). The bootstrapped heap example is a standard test of recursive modules for ML, and it is described by Russo [2001], Leroy [2003], and Dreyer [2005a].

Figure 2.13 shows how bootstrapped heaps can be defined with units. The *Belem* unit corresponds to the `BootstrappedElem` structure where the *elemPT* import corresponds to `Elem.T`, the *compareP* import to `Elem.compare`, and the *heapT* import to `PrimH.Heap`. The **compound** expression in the definition of *Bootstrap* combines a unit *MakeH* that builds primitive heaps with the *Belem* unit to create the core of the bootstrapped heap. The surrounding unit encapsulates the bootstrapped heap with the interface of a heap-making unit. It imports an element and exports a bootstrapped heap of those elements. The function abstracts this unit over the primitive heap-making unit to be used in the construction of the bootstrapped heap.

[8]Bootstrapped heaps support constant time `findMin`, `merge`, and `insert` operations, and a logarithmic `deleteMin` operation. In keeping with the theme of Okasaki's book, they are a persistent (i.e., purely functional) data structure.

```
τ = unitT [elemT, compare:elemT→elemT→int, heapT,
              insert:elemT→heapT→heapT]
              (elemT Compare → heapT insert)

Belem =
   unit import [elemPT, compareP:elemPT→elemPT→int, heapT]
          export [elemT=int+(elemPT×heapT), compare:elemT→elemT→int].
      elemT=int + (elemPT × heapT)
      compare:elemT→elemT→int = λ e:elemT. λ e:elemT. ...compareP...

Bootstrap : τ→τ =
   λ MakeH : τ.
      unit import [elemT⁰, compare⁰:elemT⁰→elemT⁰→int]
             export [heapT⁰, insert:elemT⁰→heapT⁰→heapT⁰].
         invoke
            (compound import []
                            export [heapT¹, elemT²=int+(elemT⁰×heapT¹),
                                        insert:elemT²→heapT¹→heapT¹]
               link MakeH:import [elemT¹, compare¹:elemT¹→elemT¹→int]
                              export [heapT¹, insert¹:elemT¹→heapT¹→heapT¹]
                      Belem:import [elemPT², compareP²:elemPT²→elemPT²→int,
                                         heapT²]
                             export [elemT²=int+(elemPT²×heapT²),
                                         compare²:elemT²→elemT²→int]
               where (elemT¹←elemT²) (compare¹←compare²)
                        (elemPT²←elemT⁰) (compareP²←compare⁰) (heapT²←heapT¹)
            ) as PrimH:...
         heapT = PrimH.heapT
         insert:PrimH.elemT→PrimH.heapT→PrimH.heapT = ...PrimH.insert...
```

Figure 2.13. Recursive linking example: bootstrapped heap

2.3.3.3 Independent compilation The ML-based bootstrapped heap example above is made challenging by the requirement that the contents of `MakeH` are not known when attempting to compile the recursive structure statement. If they were known, the compiler could merge the bodies of the two structures (`BootstrappedElem` and `PrimH`) and account for recursive dependencies directly with ML's facilities for recursive type and value definitions. However, such a restriction would destroy independent compilation because the application of functors such as `MakeH` would rely entirely on the functor's implementation, instead of just its interface.

The following recursive structure definition typifies a more difficult situation where both parts of a recursive structure definition are placed in separate components.

```
structure rec S = Sfun(T)
and T = Tfun(S)
```

The `Sfun` functor consumes a `T` structure and produces an `S` structure and `Tfun` consumes an `S` to produce a `T`.

Although the recursive structure can be type checked and compiled, a run-time error occurs because the `T` structure is used before its definition is executed. The error arises even if the functor bodies consist entirely of syntactic values, because it arises from an ill-founded recursion at the structure-level. Russo [2001] solves the problem by using eta-expanded structure definitions as the functors' parameters. Essentially, the production of a structure from a functor requires evaluation of the functor's body which refers to the as-of-yet undefined argument; the eta expanded argument structure protects all references to the original argument structure with lambda bindings, thereby delaying references to the argument until after the entire recursive structure is built (in fact until the function is actually invoked). In contrast, unit compounding does not execute the bodies of any units, and so there is so there is no need to delay the reference to an import.

2.4 Toward a Practical Language

The model of modules and units presented thus far forms the basis of an ML-like component-oriented programming language. The following additions to the model would make it a practical language. Chapters 4 and 5 discuss many of these concerns in the setting of an untyped language.

Signatures: Signature definitions enable reuse of import and export specifications for units and compound units. An ML-like signature facility would work with little modification, as long as a **where** clause for an import signature can refer to bindings in the export signatures, and *vice versa* (e.g., recursive dependent signatures [Crary et al. 1999]).

Type annotations: The type annotations on value definitions and function parameters should be optional and supplied by type inference. The context annotation on **invoke** and the import/export annotation on each subpart of a compound unit (that is, in the U production) could be relaxed to simple lists of imported and exported identifiers, or they could be supplied as signatures. However, these annotations should not be omitted entirely since they specify which names are introduced by the **invoke**, and which names are available as linkages in a **compound**.

Because component-oriented programming requires full documentation of component interfaces, there should be no attempt to infer the imports and exports (and hence types) of unit values. Thus, the main difficulty in performing type inference is the addition of subtyping to the core language. This difficulty could be easily avoided by making units second-class, and hence completely trackable at compile time.

Linking: Specifying a linkage for each import in a compound is tedious in all but the smallest systems. Linkages should be specified between signatures, which would denote linkages of all of their components.

Renaming: Our model uses stamps to differentiate between internal and external names. A practical system should provide mechanisms for explicitly managing the mapping between internal and external names on unit imports and exports.

Compilation management: Compilation management in this setting entails searching the pool of top-level modules to find and compile each of a module's dependencies before compiling the module itself. To make the search unambiguous, each module in the pool must have a unique name, although in a practical setting uniqueness might instead be required at the level of file paths or URIs. This is most similar to Objective Caml's [Leroy 2004] compilation management, which makes all definitions (excepting in the interactive environment) appear in a top-level structure whose name corresponds with the name of the file containing it. (Top-level structures can be sealed with signatures and used as functor arguments, just like other structures.)

Haskell's and PLT Scheme's compilation management are also based on top-level modules; however, dependencies are stated explicitly by importing entire modules, instead of implicitly as the heads of paths.

Top-level components: Often, when writing highly componentized programs, top-level modules contain only the definition of a single unit. Direct support for this idiom with a top-level component construct that combines unit and module features could be implemented with a combination of our modules and units.

2.5 Related Work

Dreyer et al. [2003] provide a definitive model of ML module systems, but they do not include support for cyclic dependencies. Type-theoretic work on cyclic dependencies for ML has yielded the notion of recursive dependent signatures [Crary et al. 1999] and an account of recursive type generativity that resembles the backpatching semantics of Scheme recursion [Dreyer 2005a]. Russo adapts the former to a practical extension of SML with recursive structures [Russo 2001]; since functor linking remains tied to functor invocation in this system, mutually-recursive functions across functor boundaries work only with an eta expansion, as discussed in Section 2.3.3.

Dreyer's dissertation [Dreyer 2005b] provides a critique of units in comparison to ML modules and approaches to recursive functors. His comparison notes the problems with units that we address here with the first-order module construct and translucent types, and he also notes that units naturally avoid the "double-vision" problem. His dissertation also contains a recursive module system that solves these problems, but does not support separate compilation.

Harper and Lillibridge [1994] introduced translucent sums at the same time as Leroy's manifest types [Leroy 1994]. These systems introduced the kind of translucency added to units in this paper, but neither system supports recursive linking. Harper's system supports first-class functors that obey a phase distinction allowing type checking to fully precede evaluation. Units are first-class values in the same way as these functors.

2.6 Conclusion

The model of units and modules presented in this chapter is unique in supporting the following criteria: translucent and opaque type imports and exports, independent unit compilation, and recursive unit linking that avoids ill-founded recursive module definitions without requiring manual intervention (although recursive value definitions can be ill-founded).

The model accounts for internal linking and external linking as orthogonal concepts. Although external linking offers the greatest flexibility for code reuse, programming without any internal linking is infeasible. For example, the *fully functorized* style of ML programming, which requires external linkages that are performed by functor application, enforces a high overhead on the programmer in cases where parameterization is unnecessary. Furthermore, the system that manages the externally linked entities must support direct references and internal linking, so that the programmer can refer to the particular components that he wishes to externally link.

CHAPTER 3

THE FORMAL SEMANTICS OF TYPED UNITS

The language of modules and units (Figure 2.1) is formally specified with a type system and small-step operational semantics. A type soundness proof in the style of Wright and Felleisen [1994] characterizes the correspondence between the type system and operational semantics (Section 3.3).

The type system (Section 3.1) is based on the manifest type system for higher-order functors [Leroy 1994; Leroy 1996] and is similar to the structure and functor type system of Figure 2.9. Notable extensions of these systems (besides the units themselves) include support for subtyping in the base language due to first-class unit values, and support for modules that can restrict name visibility without providing sealing.

The operational semantics (Section 3.2) is specified as an evaluation-context-based term rewriting system. Such systems typically rely on substitution, but this system instead keeps all definitions in place to avoid duplicating any generative type definitions. Such duplication would render the standard proof techniques for type soundness inapplicable, since intermediate steps in a program's evaluation could become ill-typed. Additionally, the operational semantics requires a language construct (**rec**) for arbitrary recursive definitions to express the reduction of recursively linked compound units.

3.1 Type System

Table 3.1 summarizes the functions and relations that make up the type system for the language of modules and units. Figure 3.1 contains the type system.

The relations that define well-formed module descriptions ($\mathcal{M}$), bindings (B), and types (Ty) ensure that no identifier shadows another, that each type path refers to a defined type, and that module bindings and unit types are well-formed. The first restriction is for the type system's convenience; any otherwise well-formed type can meet

Table 3.1. Glossary for the module and unit type system

Relation	Meaning
$\textsc{distinct}\ \mathcal{M}$	no names are multiply defined
$\textsc{distinct}\mathrm{I}\ \mathcal{M}$	no identifiers are multiply defined
$\mathcal{M}\ \textsc{perm}\ \mathcal{M}$	one module description permutes the other
$\mathit{ids}\ \textsc{perm}\ \mathit{ids}$	one identifier sequence permutes the other
$\mathcal{M};\mathcal{M}\ \textsc{interleave}\ \mathcal{M}$	the left descriptions interleave to form the right one
$B \in \Gamma \cup \mathcal{M}$	the binding is in the context or description
$\Gamma \vdash P \mapsto B$	lookup of a path in a typing context
$\Gamma \vdash \mathcal{M}$	well-formed module descriptions
$\Gamma \vdash B$	well-formed bindings
$\Gamma \vdash \mathit{Ty}$	well-formed types
$\Gamma \vdash \mathcal{M} <: \mathcal{M}$	subtyping for module descriptions
$\Gamma \vdash B <: B$	subtyping for bindings
$\Gamma \vdash \mathit{Ty} <: \mathit{Ty}$	subtyping
$\Gamma;\mathcal{M} \vdash \mathit{Ds} : \mathcal{M}$	typing for sequences of definitions
$\Gamma \vdash D : B$	typing for definitions
$\Gamma \vdash E : \mathit{Ty}$	typing for expressions
$\Gamma \vdash_{\mathrm{P}} E : \mathit{Ty}$	as above, expands types that are paths
$\Gamma \vdash \textbf{import}\ \mathcal{M}\ \textbf{export}\ \mathcal{M} : \mathit{Ty}$	typing for unit interfaces
$\Gamma \vdash U \ldots U : \mathcal{M};\mathcal{M}$	typing for interface annotated unit expressions
$\Gamma;\mathcal{M};\mathcal{M};L \ldots L \vdash L$	well-formed unit linkages
$\vdash \mathcal{M} <: \mathcal{M}$	subtyping for module descrs. (no type expansion)
$\vdash B <: B$	subtyping for bindings (no type expansion)
$\vdash \mathit{Ty} <: \mathit{Ty}$	subtyping (no type expansion)
$\Gamma \vdash \mathcal{M}\ \delta\ \mathcal{M}$	type expansion for module descriptions
$\Gamma \vdash B\ \delta\ B$	type expansion for bindings
$\Gamma \vdash \mathit{Ty}\ \delta\ \mathit{Ty}$	type expansion
$\Gamma \vdash \mathit{Ty}\ \delta_P\ \mathit{Ty}$	type expansion for top-level variables only
Function	**Meaning**
$\textsc{dom} : \Gamma \cup \mathcal{M} \cup B \rightarrow \mathit{ids}$	gets the identifiers bound by the given bindings
$\textsc{dom}\mathrm{N} : \Gamma \cup \mathcal{M} \cup B \rightarrow \mathit{ns}$	gets the names bound the given bindings
$\textsc{Lleft} : L \ldots L \rightarrow \mathit{ids}$	gets the left side of a linkage
$\widehat{P} : P$	puts hats on the names in the path

$$\boxed{\Gamma \vdash \mathcal{M}}$$

$$\frac{}{\Gamma \vdash \epsilon}\ (\mathcal{M}\epsilon) \qquad \frac{\Gamma \vdash B \qquad \Gamma, B \vdash \mathcal{M} \qquad \textsc{dom}(B) \cap \textsc{dom}(\Gamma) = \emptyset}{\Gamma \vdash B, \mathcal{M}}\ (\mathcal{M})$$

$$\boxed{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash Ty}{\Gamma \vdash x^i \mathbf{:} Ty}\ (\textsc{Bval}) \qquad \frac{}{\Gamma \vdash t^i}\ (\textsc{Btype1}) \qquad \frac{\Gamma \vdash Ty}{\Gamma \vdash t^i \mathbf{=} Ty}\ (\textsc{Btype2})$$

$$\frac{\Gamma \vdash \mathcal{M} \qquad ns \subseteq \textsc{dom}\mathrm{N}(\mathcal{M}) \qquad \textsc{distinct}(\mathcal{M})}{\Gamma \vdash m^i \mathbf{:} \mathcal{M}\ \mathbf{(}ns\mathbf{)}}\ (\textsc{Bmod})$$

$$\boxed{\Gamma \vdash Ty}$$

$$\frac{}{\Gamma \vdash \mathbf{int}}\ (\textsc{Tint}) \qquad \frac{\Gamma \vdash Ty_1 \qquad \Gamma \vdash Ty_2}{\Gamma \vdash Ty_1 \to Ty_2}\ (\textsc{Tfun}) \qquad \frac{\Gamma \vdash Ty_1 \qquad \Gamma \vdash Ty_2}{\Gamma \vdash Ty_1 + Ty_2}\ (\textsc{Tsum})$$

$$\frac{\Gamma \vdash Ty_1 \qquad \Gamma \vdash Ty_2}{\Gamma \vdash Ty_1 \times Ty_2}\ (\textsc{Tprod}) \qquad \frac{(\Gamma \vdash P \mapsto t^i) \vee (\Gamma \vdash P \mapsto t^i \mathbf{=} Ty)}{\Gamma \vdash P}\ (\textsc{Tpath})$$

$$\frac{\Gamma \vdash \mathcal{M} \qquad ns_1 \cup ns_2 = \textsc{dom}\mathrm{N}(\mathcal{M}) \qquad ns_1 \cap ns_2 = \emptyset \qquad \textsc{distinct}(\mathcal{M})}{\Gamma \vdash \mathbf{unitT}\ \mathcal{M}\ \mathbf{(}ns_1 \to ns_2\mathbf{)}}\ (\textsc{Tunit})$$

Figure 3.1. Type system for modules and units

$$\boxed{\Gamma; \mathcal{M} \vdash \mathit{Ds} : \mathcal{M}}$$

$$\frac{}{\Gamma; \epsilon \vdash \epsilon : \epsilon}\ (\textsc{Ds}\epsilon)$$

$$\frac{\Gamma \vdash D : B \qquad (\Gamma\textbf{,}B); \mathcal{M}_1 \vdash \mathit{Ds} : \mathcal{M}_2 \qquad \textsc{dom}(B) \cap \textsc{dom}(\Gamma) = \emptyset}{\Gamma; \mathcal{M}_1 \vdash (D\ \mathit{Ds}) : (B\textbf{,}\mathcal{M}_2)}\ (\textsc{Ds}1)$$

$$\frac{\Gamma \vdash B \qquad (\Gamma\textbf{,}B); \mathcal{M}_1 \vdash \mathit{Ds} : \mathcal{M}_2 \qquad \textsc{dom}(B) \cap \textsc{dom}(\Gamma) = \emptyset}{\Gamma; (B\textbf{,}\mathcal{M}_1) \vdash \mathit{Ds} : (B\textbf{,}\mathcal{M}_2)}\ (\textsc{Ds}2)$$

$$\boxed{\Gamma \vdash D : B}$$

$$\frac{\Gamma \vdash \mathit{Ty}_1 \qquad \Gamma \vdash E : \mathit{Ty}_2 \qquad \Gamma \vdash \mathit{Ty}_2 <: \mathit{Ty}_1}{\Gamma \vdash (x^i\textbf{:}\mathit{Ty}_1\textbf{=}E) : (x^i\textbf{:}\mathit{Ty}_1)}\ (\textsc{Dval}) \qquad \frac{\Gamma \vdash \mathit{Ty}}{\Gamma \vdash (t^i\textbf{=}\mathit{Ty}) : (t^i\textbf{=}\mathit{Ty})}\ (\textsc{Dtype})$$

$$\frac{\begin{array}{c}\Gamma \vdash E : \mathit{Ty}_1 \\ \mathit{Ty}_2 = \textbf{unitT}\ \mathcal{M}\ \textbf{(}\epsilon \rightarrow \textsc{dom}\mathrm{N}(\mathcal{M})\textbf{)} \qquad \Gamma \vdash \mathit{Ty}_2 \qquad \Gamma \vdash \mathit{Ty}_1 <: \mathit{Ty}_2\end{array}}{\Gamma \vdash (\textbf{invoke}\ E\ \textbf{as}\ m^i\textbf{:}\mathcal{M}) : (m^i\textbf{:}\mathcal{M}\ \textbf{(}\textsc{dom}\mathrm{N}(\mathcal{M})\textbf{)})}\ (\textsc{Dinv})$$

$$\frac{\Gamma; \epsilon \vdash \mathit{Ds} : \mathcal{M} \qquad \mathit{ns} \subseteq \textsc{dom}\mathrm{N}(\mathcal{M}) \qquad \textsc{distinct}(\mathcal{M})}{\Gamma \vdash (\textbf{module}\ m^i\ \textbf{provide}\ \mathit{ns}\textbf{=}\mathit{Ds}) : (m^i\textbf{:}\mathcal{M}\ \textbf{(}\mathit{ns}\textbf{)})}\ (\textsc{Dmod}1)$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{M}_1 \\ \textsc{distinct}(\mathcal{M}_1) \qquad \mathit{ns} \subseteq \textsc{dom}\mathrm{N}(\mathcal{M}_1) \qquad \Gamma; \epsilon \vdash \mathit{Ds} : \mathcal{M}_2 \qquad \Gamma \vdash \mathcal{M}_2 <: \mathcal{M}_1\end{array}}{\Gamma \vdash (\textbf{module}\ m^i\textbf{:}\mathcal{M}_1\ \textbf{provide}\ \mathit{ns}\textbf{=}\mathit{Ds}) : (m^i\textbf{:}\mathcal{M}_1\ \textbf{(}\mathit{ns}\textbf{)})}\ (\textsc{Dmod}2)$$

Figure 3.1. continued

$$\boxed{\Gamma \vdash E : \mathit{Ty}}$$

$$\frac{z \in \mathbb{Z}}{\Gamma \vdash z : \textbf{int}}\ (\textsc{Eint}) \qquad \frac{\Gamma \vdash P_{\mathrm{x}} \mapsto \mathrm{x}^i\mathbf{:}\mathit{Ty}}{\Gamma \vdash P_{\mathrm{x}} : \mathit{Ty}}\ (\textsc{Epath})$$

$$\frac{\Gamma \vdash E : \mathit{Ty}_1 \qquad \Gamma \vdash \mathit{Ty}_2}{\Gamma \vdash \textbf{injl}\ E : \mathit{Ty}_1 + \mathit{Ty}_2}\ (\textsc{Einjl}) \qquad \frac{\Gamma \vdash E : \mathit{Ty}_2 \qquad \Gamma \vdash \mathit{Ty}_1}{\Gamma \vdash \textbf{injr}\ E : \mathit{Ty}_1 + \mathit{Ty}_2}\ (\textsc{Einjr})$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{P}} E_1 : \mathit{Ty}_1 + \mathit{Ty}_2 \\ \Gamma \vdash_{\mathrm{P}} E_2 : \mathit{Ty}_3 \rightarrow \mathit{Ty}_4 \qquad \Gamma \vdash_{\mathrm{P}} E_3 : \mathit{Ty}_5 \rightarrow \mathit{Ty}_6 \qquad \Gamma \vdash \mathit{Ty}_1 <: \mathit{Ty}_3 \\ \Gamma \vdash \mathit{Ty}_2 <: \mathit{Ty}_5 \qquad \Gamma \vdash \mathit{Ty}_4 <: \mathit{Ty}_7 \qquad \Gamma \vdash \mathit{Ty}_6 <: \mathit{Ty}_7 \qquad \Gamma \vdash \mathit{Ty}_7\end{array}}{\Gamma \vdash \textbf{case}\ E_1\ \textbf{of}\ E_2 + E_3 : \mathit{Ty}_7}\ (\textsc{Ecase})$$

$$\frac{\Gamma \vdash E_1 : \mathit{Ty}_1 \qquad \Gamma \vdash E_2 : \mathit{Ty}_2}{\Gamma \vdash \mathbf{(}E_1\mathbf{,}\ E_2\mathbf{)} : \mathit{Ty}_1 \times \mathit{Ty}_2}\ (\textsc{Eprod})$$

$$\frac{\Gamma \vdash_{\mathrm{P}} E : \mathit{Ty}_1 \times \mathit{Ty}_2}{\Gamma \vdash \pi_1 E : \mathit{Ty}_1}\ (\textsc{Epj1}) \qquad \frac{\Gamma \vdash_{\mathrm{P}} E : \mathit{Ty}_1 \times \mathit{Ty}_2}{\Gamma \vdash \pi_2 E : \mathit{Ty}_2}\ (\textsc{Epj2})$$

$$\frac{\begin{array}{c}\mathrm{x}^i \notin \textsc{dom}(\Gamma) \qquad \Gamma \vdash \mathit{Ty}_1 \\ \Gamma\mathbf{,}(\mathrm{x}^i\mathbf{:}\mathit{Ty}_1) \vdash E : \mathit{Ty}_2\end{array}}{\Gamma \vdash (\lambda \mathrm{x}^i\mathbf{:}\mathit{Ty}_1\mathbf{.}E) : (\mathit{Ty}_1 \rightarrow \mathit{Ty}_2)}\ (\textsc{Efun}) \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{P}} E_1 : \mathit{Ty}_1 \rightarrow \mathit{Ty}_2 \\ \Gamma \vdash E_2 : \mathit{Ty}_3 \qquad \Gamma \vdash \mathit{Ty}_3 <: \mathit{Ty}_1\end{array}}{\Gamma \vdash E_1\ E_2 : \mathit{Ty}_2}\ (\textsc{Eapp})$$

$$\frac{\Gamma \vdash \mathit{Ds} : \mathcal{M} \qquad \Gamma\mathbf{,}\mathcal{M} \vdash E : \mathit{Ty}_2 \qquad \Gamma\mathbf{,}\mathcal{M} \vdash \mathit{Ty}_2 <: \mathit{Ty}_1 \qquad \Gamma \vdash \mathit{Ty}_1}{\Gamma \vdash \textbf{let}\ \mathit{Ds}\ \textbf{in}\ E : \mathit{Ty}_1}\ (\textsc{Elet})$$

$$\frac{\begin{array}{c}\Gamma \vdash \textbf{import}\ \mathcal{M}_1\ \textbf{export}\ \mathcal{M}_2 : \mathit{Ty} \\ \mathit{Ty} = \textbf{unitT}\ \mathcal{M}_4\ \mathbf{(}\mathit{ns}_1 \rightarrow \mathit{ns}_2\mathbf{)} \qquad \Gamma; \mathcal{M}_1 \vdash \mathit{Ds} : \mathcal{M}_3 \qquad \Gamma \vdash \mathcal{M}_3 <: \mathcal{M}_4\end{array}}{\Gamma \vdash \textbf{unit import}\ \mathcal{M}_1\ \textbf{export}\ \mathcal{M}_2\mathbf{.}\ \mathit{Ds} : \mathit{Ty}}\ (\textsc{Eunit})$$

$$\frac{\begin{array}{c}\Gamma \vdash \textbf{import}\ \mathcal{M}_1\ \textbf{export}\ \mathcal{M}_2 : \mathit{Ty} \\ \mathit{Ty} = \textbf{unitT}\ \mathcal{M}_7\ \mathbf{(}\mathit{ns}_1 \rightarrow \mathit{ns}_2\mathbf{)} \qquad \Gamma \vdash U_1 \ldots U_m : \mathcal{M}_3; \mathcal{M}_4 \\ \textsc{distinct}\mathrm{I}\ (\mathcal{M}_1\mathbf{,}\mathcal{M}_3\mathbf{,}\mathcal{M}_4) \qquad \textsc{perm}\ (\mathrm{L}\textsc{left}(L_1 \ldots L_n))\ (\textsc{dom}(\mathcal{M}_3)) \\ \mathcal{M}_5 = \mathcal{M}_4\{\mathit{id}_1 \leftarrow \mathit{id}_2 \mid \exists i.\, L_i = (\mathit{id}_1 \leftarrow \mathit{id}_2)\} \qquad \textsc{perm}\ (\mathcal{M}_1\mathbf{,}\mathcal{M}_5)\ \mathcal{M}_6 \\ \Gamma \vdash \mathcal{M}_6 \qquad \forall i \leq n\,.\ \Gamma; \mathcal{M}_3; \mathcal{M}_6; (L_1 \ldots L_n) \vdash L_i \qquad \Gamma \vdash \mathcal{M}_6 <: \mathcal{M}_7\end{array}}{\begin{array}{c}\Gamma \vdash \textbf{compound import}\ \mathcal{M}_1\ \textbf{export}\ \mathcal{M}_2\ \textbf{link}\ U_1 \ldots U_m\ \textbf{where}\ L_1 \ldots L_n : \\ \mathit{Ty}\end{array}}\ (\textsc{Ecmpd})$$

Figure 3.1. continued

$$\boxed{\Gamma \vdash \textbf{import } \mathcal{M} \textbf{ export } \mathcal{M} : \mathit{Ty}}$$

$$\frac{\begin{array}{c}\mathcal{M}_1 ; \mathcal{M}_2 \text{ INTERLEAVE } \mathcal{M}_3 \\ \mathit{Ty} = \textbf{unitT } \mathcal{M}_3 \ \textbf{(}\text{DOMN}(\mathcal{M}_1) \rightarrow \text{DOMN}(\mathcal{M}_2)\textbf{)} \qquad \Gamma \vdash \mathit{Ty}\end{array}}{\Gamma \vdash \textbf{import } \mathcal{M}_1 \textbf{ export } \mathcal{M}_2 : \mathit{Ty}} \ (\text{IE})$$

$$\boxed{\Gamma \vdash U \ldots U : \mathcal{M} ; \mathcal{M}}$$

$$\frac{}{\Gamma \vdash \epsilon : \epsilon ; \epsilon} \ (\text{U}\epsilon)$$

$$\frac{\begin{array}{c}\Gamma \vdash E : \mathit{Ty}_1 \qquad \Gamma \vdash \textbf{import } \mathcal{M}_1 \textbf{ export } \mathcal{M}_2 : \mathit{Ty}_2 \\ \Gamma \vdash \mathit{Ty}_1 <: \mathit{Ty}_2 \qquad \Gamma \vdash U_2 \ldots U_m : \mathcal{M}_3 ; \mathcal{M}_4\end{array}}{\Gamma \vdash (E\textbf{:import } \mathcal{M}_1 \textbf{ export } \mathcal{M}_2) \ U_2 \ldots U_m : \mathcal{M}_1\textbf{,}\mathcal{M}_3 ; \mathcal{M}_2\textbf{,}\mathcal{M}_4} \ (\text{U})$$

$$\boxed{\Gamma ; \mathcal{M} ; \mathcal{M} ; L \ldots L \vdash L}$$

$$\frac{\begin{array}{c}B_1 \in \mathcal{M}_1 \qquad \text{DOM}(B_1) = \mathit{id}_1 \qquad B_2 \in \mathcal{M}_2 \qquad \text{DOM}(B_2) = \mathit{id}_2 \\ B_3 = B_1\{\mathit{id}_1 \leftarrow \mathit{id}_2 \mid \exists i.\ L_i = (\mathit{id}_1 \leftarrow \mathit{id}_2)\} \qquad \Gamma\textbf{,}\mathcal{M}_2 \vdash B_2 <: B_3\end{array}}{\Gamma ; \mathcal{M}_1 ; \mathcal{M}_2 ; L_1 \ldots L_m \vdash \mathit{id}_1 \leftarrow \mathit{id}_2} \ (\text{L})$$

$$\boxed{\Gamma \vdash_{\text{P}} E : \mathit{Ty}}$$

$$\frac{\Gamma \vdash E : \mathit{Ty}_1 \qquad \Gamma \vdash \mathit{Ty}_1 \ \delta^{\text{P}} \ \mathit{Ty}_2 \qquad \mathit{Ty}_2 \text{ is not a path } (P)}{\Gamma \vdash_{\text{P}} E : \mathit{Ty}_2}$$

Figure 3.1. continued

these restrictions with α-renaming of the stamps on identifiers. The rule for module bindings (Bmod) ensures that each provided name is bound by the module, and that no name is multiply bound in a module's description. The rule for units (Tunit) establishes that the lists of imported and exported names exactly partition the unit's import/export description into imported and exported bindings. The import description is not separated from the export description because an import can refer to an exported type and vice versa.

The type rules for definitions produce bindings, and the type rules for definition sequences produce descriptions. The definition sequence rules build the description and ensure that no identifier shadows another. They have an additional context argument that the Ds2 rule uses as a source for bindings; it is ϵ except when type checking the body of a unit, as discussed with units below. The Dval and Dtype rules produce the specified type, and ensure that the resulting binding is well-formed. The Dinv rule checks that its subexpression has a unit type with no imports. The expression's type must be a subtype of the specified context, suitably placed into a unit type. The Dmod1 and Dmod2 rules ensure that the provided names are defined, and that no externally visible names are multiply defined. Additionally, Dmod2 ensures that the body of a sealed module is consistent with the sealing description. The body of a sealed module can define a name multiple times; since only the names in the explicitly given description are visible from outside the module, no ambiguity can arise.

The type system for expressions is typical of typed λ-calculi, but with the addition of the Eunit and Ecmpd rules. The IE rule builds a unit's type from its declared imports and exports, and it uses the Tunit rule to ensure that the resulting type is well-formed; in particular, the same name cannot be both imported and exported. Interleaving allows imports to depend upon types declared in the export portion by placing the imported type after the exported one. The Eunit rule checks the body of the unit, using the unit's declared imports as the second argument to the declaration rule. The Ds2 rule then intersperses the imports as needed while checking the unit's body.

The following example demonstrates typechecking for units.

```
module unit_example provide u_ex =
  u_ex : unitT [t, v:t, w:t] (v → w t) =
    unit import [v:t] export [t, w:t→t].
      t = int
      w:t→t = λ x:t. x + v
```

The type of import v refers to the exported type t. When typechecking the body of the unit, the binding for v must be inserted after checking the definition of t, but before the definition of w. In general, each imported binding should be inserted at the earliest point where it is well-defined to do so.[1]

The ECMPD rule also uses IE to build the result type. However, the process for creating the description for the compound unit's "body", $\mathcal{M}_6$, is more involved than for noncompound units. The U check ensures that the type of each subunit expression is compatible with the corresponding import and export descriptions. The DISTINCTI check ensures that there is no ambiguity in what each linkage refers to, and the check involving LIMPORTS ensures that each subunit import is linked to exactly once.

The substitution applies the linkages to the subunits' exports, redirecting type and module references from a subunit's import to the exported definition that satisfies the import. The L rule ensures that the type of a linkage's export is a subtype of its import. Since the type of an import can refer to types and modules defined in other imports (of the same subunit), the L rule performs linkage substitution on the import before comparing it to the corresponding export (for the subtyping check to possibly succeed, the substitution must rename the identifier of B_1).

The following example (which includes the identifier stamps), demonstrates typechecking for compound units.

```
module compound_example provide =
  $u^1$ : unitT [$t^2$, $w^1$:int→$t^2$] ($t$ → $w$) =
    unit import [$t^2$] export [$w^1$:int→$t^2$]. ...
  $u^2$ : unitT [$u^4$=int] ( → $u$) =
    unit import [] export [$u^4$=int]. ...
  $u^3$ : unitT [$w^1$:int→int] ( → $w$) =
    compound import [] export [$w^1$:int→int].
      link $u^2$ : import [] export [$u^4$=int]
           $u^1$ : import [$t^2$] export [$w^1$:int→$t^2$]
      where $t^2$ ← $u^4$
```

The intermediate context $\mathcal{M}_4$ is [u^4**=int,** w^1**:int** $\rightarrow$ t^2], so that the context after substitution is $\mathcal{M}_5 =$ [u^4**=int,** w^1**:int** $\rightarrow$ u^4].

[1]Modules provide a convenient mechanism for grouping imports and exports. However, they have limited flexibility for recursive dependencies, because the entire module binding must appear atomically, which limits interleaving. In the example, if t and w were grouped in a module, v could not be interspersed. This restriction could be relaxed by letting the unit rule split the module body's context throughout its enclosing context, or by using a signature mechanism as mentioned in Section 2.4 (instead of modules) to group imports and exports.

The permutation of $\mathcal{M}_1$,$\mathcal{M}_5$ allows the subunits to have mutual type dependencies while stopping short of supporting a true recursive type environment, and equi-recursive types. In the following interpreter-inspired example, suppose that the types of the two *run* functions refer to the *dec* and *exp* types, and that these two types depend on each other. Then in $\mathcal{M}_6$, the bindings for exp^2 and dec^1 must both precede the bindings for *runD* and *runE*.

```
exp_u:... = unit import [dec,runD:...] export [exp,runE:...]. ...
dec_u:... = unit import [exp,runE:...] export [dec,runD:...]. ...
run_u:... =
  compound import [] export [dec², exp¹, runE¹:..., runD²:...]
    link exp_u:import [dec¹, runD¹:...] export [exp¹, runE¹:...]
        dec_u:import [exp²,runE²:...] export [dec²,runD²:...]
    where (dec¹←dec²) (runD¹←runD²)
          (exp²←exp¹) (runE²←runE¹)
```

If the *exp* and *dec* types were exported translucently, exposing the mutual reference, no ordering would allow *run* to typecheck, since *exp* would need to precede *dec* and simultaneously, *exp* would need to precede *dec*.

3.1.1 Path Lookup

When looking up a path in a context (Figure 3.2), any types in the returned binding are lifted out of their scope. Paths that are contained within the type itself, and that reference bindings in their enclosing module, would then become free references, or get captured by an intervening binding. The substitution prevents this by changing these identifiers into paths to their original binding. The substitution must use hatted names in the path, so the reference can succeed even if the name is not provided.

For example, with the following module definition the path $m^1.n.v$ has type $m^1.\widehat{t} \rightarrow m^1.\widehat{n}.\widehat{u}$, which is equal to **int** $\rightarrow$ **int**.

```
module m¹ provide n =
  t² = int
  module n³ provide v =
    u⁴ = int
    v⁵:t²→u⁴ = ...
```

Without the hatted identifiers, the result type would be $m^1.t \rightarrow m^1.n.u$ which refers to names that are not provided by the module, and are hence inaccessible through the LOOK2 rule.

$$\boxed{\widehat{P_m} = P_m}$$

$$\begin{array}{rcl} \widehat{m^i} & = & m^i \\ \widehat{P_m.m} & = & \widehat{P_m}.\widehat{m} \\ \widehat{P_m.\widehat{m}} & = & \widehat{P_m}.\widehat{m} \end{array}$$

$$\boxed{\Gamma \vdash P \mapsto B}$$

$$\frac{B \in \Gamma \qquad \textsc{dom}(B) = id}{\Gamma \vdash id \mapsto B}\ (\textsc{look}1)$$

$$\frac{\Gamma \vdash P \mapsto m^i{:}\mathcal{M}\ (ns) \qquad B \in \mathcal{M} \qquad \textsc{dom}\mathrm{N}(B) = n_1 \qquad n_1 \in ns}{\Gamma \vdash P.n_1 \mapsto B\{n_2^j \leftarrow \widehat{P.n_2} \mid n_2^j \in \textsc{dom}(\mathcal{M})\}}\ (\textsc{look}2)$$

$$\frac{\Gamma \vdash P \mapsto m^i{:}\mathcal{M}\ (ns) \qquad B \in \mathcal{M} \qquad \textsc{dom}\mathrm{N}(B) = n_1}{\Gamma \vdash P.\widehat{n_1} \mapsto B\{n_2^j \leftarrow \widehat{P.n_2} \mid n_2^j \in \textsc{dom}(\mathcal{M})\}}\ (\textsc{look}3)$$

Figure 3.2. Path lookup

3.1.2 Subtyping

Subtyping is based on a simple structural subtyping relation (Figure 3.3). A description $\mathcal{M}_1$ is a subtype of $\mathcal{M}_2$ if each identifier bound in $\mathcal{M}_2$ appears in $\mathcal{M}_1$, bound to a subbinding. Binding subtypes make a transparent type binding a subtype of an opaque type binding; however two transparent type bindings must have equal types. This is because references to the transparent type could occur in both co- and contra-variant positions. Subtyping for unit types is similar to context subtyping, but based on the defined names instead of identifiers. Unit subtyping is contra-variant in the imports and co-variant in the exports.

Type paths (P_t) are structural subtypes only if they are equal, but the general

$$\boxed{Ty <: Ty}$$

$$\frac{}{\mathbf{int} <: \mathbf{int}}\ (\textsc{sub}\mathrm{T}\textsc{y}1) \qquad \frac{\widehat{P_{t1}} = \widehat{P_{t2}}}{P_{t1} <: P_{t2}}\ (\textsc{sub}\mathrm{T}\textsc{y}2)$$

$$\frac{Ty_3 <: Ty_1 \qquad Ty_2 <: Ty_4}{Ty_1 \to Ty_2 <: Ty_3 \to Ty_4}\ (\textsc{sub}\mathrm{T}\textsc{y}3)$$

$$\frac{Ty_1 <: Ty_3 \qquad Ty_2 <: Ty_4}{Ty_1 + Ty_2 <: Ty_3 + Ty_4}\ (\textsc{sub}\mathrm{T}\textsc{y}4) \qquad \frac{Ty_1 <: Ty_3 \qquad Ty_2 <: Ty_4}{Ty_1 \times Ty_2 <: Ty_3 \times Ty_4}\ (\textsc{sub}\mathrm{T}\textsc{y}5)$$

$$\frac{\begin{array}{ll} ns_4 \subseteq ns_2 & \forall B_2 \in \mathcal{M}_2.\ (\textsc{dom}\mathrm{N}(B_2) \subseteq ns_4) \Rightarrow \exists B_1 \in \mathcal{M}_1.\ B_1 <: B_2 \\ ns_1 \subseteq ns_3 & \forall B_1 \in \mathcal{M}_1.\ (\textsc{dom}\mathrm{N}(B_1) \subseteq ns_1) \Rightarrow \exists B_2 \in \mathcal{M}_2.\ B_2 <: B_1 \end{array}}{\mathbf{unitT}\ \mathcal{M}_1\ (ns_1 \to ns_2) <: \mathbf{unitT}\ \mathcal{M}_2\ (ns_3 \to ns_4)}\ (\textsc{sub}\mathrm{T}\textsc{y}6)$$

$$\boxed{\mathcal{M} <: \mathcal{M}}$$

$$\frac{\forall B_2 \in \mathcal{M}_2.\ \exists B_1 \in \mathcal{M}_1.\ B_1 <: B_2}{\mathcal{M}_1 <: \mathcal{M}_2}\ (\textsc{sub}\mathcal{M})$$

$$\boxed{B <: B}$$

$$\frac{Ty_1 <: Ty_2}{x^i\mathbf{:}Ty_1 <: x^i\mathbf{:}Ty_2}\ (\textsc{sub}\mathrm{B}1) \qquad \frac{}{t^i <: t^i}\ (\textsc{sub}\mathrm{B}2) \qquad \frac{}{t^i\mathtt{=}Ty <: t^i}\ (\textsc{sub}\mathrm{B}3)$$

$$\frac{}{t^i\mathtt{=}Ty <: t^i\mathtt{=}Ty}\ (\textsc{sub}\mathrm{B}4) \qquad \frac{ns_2 \subseteq ns_1 \qquad \mathcal{M}_1 <: \mathcal{M}_2}{m^i\mathbf{:}\mathcal{M}_1\ (ns_1) <: m^i\mathbf{:}\mathcal{M}_2\ (ns_2)}\ (\textsc{sub}\mathrm{B}5)$$

Figure 3.3. Structural subtyping

subtyping relations first allow type paths to be replaced with their definitions, and then they check structural subtyping (Figures 3.4 and 3.5).

3.1.3 Type Checking

The typing relations defined in Figure 3.1 can be interpreted as a syntax-directed type checking algorithm that, given a context Γ, computes the type of an expression, definition, or definition sequence. There are only three exceptions to the syntax-directed nature of the rules: the choice of applying rule Ds1 or rule Ds2, the choice of the result type of the Ecase rule, and the choice of $\mathcal{M}_6$ in the Ecmpd rule. Furthermore, for type checking to be decidable, the subtyping relation must be decidable. None of these issues prevent algorithmic type checking.

An algorithm can choose between Ds1 and Ds2 by preferring Ds2 as soon as the free variables of the binding in question are defined in the context. Similarly, $\mathcal{M}_6$ can be constructed using a topological sort to ensure that every free variable in $\mathcal{M}_6$'s bindings is preceded by its definition. If there is a cycle such that no such ordering is possible, then no permutation of $\mathcal{M}_6$ is well-formed, so the **compound** expression has no type. Subtyping is decidable by first expanding all type definitions, and then by syntax-directed application of the structural subtyping rules. Similarly, the result type of a case can be determined by fully expanding the types and computing the least upper bound for the

$$\boxed{\Gamma \vdash Ty <: Ty}$$

$$\frac{\Gamma \vdash Ty_1 \;\delta\; Ty_3 \qquad \Gamma \vdash Ty_2 \;\delta\; Ty_4 \qquad Ty_3 <: Ty_4}{\Gamma \vdash Ty_1 <: Ty_2}$$

$$\boxed{\Gamma \vdash B <: B}$$

$$\frac{\Gamma \vdash B_1 \;\delta\; B_3 \qquad \Gamma \vdash B_2 \;\delta\; B_4 \qquad B_3 <: B_4}{\Gamma \vdash B_1 <: B_2}$$

$$\boxed{\Gamma \vdash \mathcal{M} <: \mathcal{M}}$$

$$\frac{\Gamma \vdash \mathcal{M}_1 \;\delta\; \mathcal{M}_3 \qquad \Gamma \vdash \mathcal{M}_2 \;\delta\; \mathcal{M}_4 \qquad \mathcal{M}_3 <: \mathcal{M}_4}{\Gamma \vdash \mathcal{M}_1 <: \mathcal{M}_2}$$

Figure 3.4. Subtyping

$$\boxed{\Gamma \vdash \mathcal{M} \;\delta\; \mathcal{M}}$$

$$\frac{}{\Gamma \vdash \epsilon \;\delta\; \epsilon}\;(\textsc{exp}\mathcal{M}\epsilon) \qquad \frac{\Gamma \vdash B_1 \;\delta\; B_2 \qquad \Gamma, B_1 \vdash \mathcal{M}_1 \;\delta\; \mathcal{M}_2}{\Gamma \vdash B_1, \mathcal{M}_1 \;\delta\; B_2, \mathcal{M}_2}\;(\textsc{exp}\mathcal{M})$$

$$\boxed{\Gamma \vdash B \;\delta\; B}$$

$$\frac{\Gamma \vdash Ty_1 \;\delta\; Ty_2}{\Gamma \vdash x^i{:}Ty_1 \;\delta\; x^i{:}Ty_2}\;(\textsc{exp}\mathrm{B1}) \qquad \frac{}{\Gamma \vdash t^i \;\delta\; t^i}\;(\textsc{exp}\mathrm{B2})$$

$$\frac{\Gamma \vdash Ty_1 \;\delta\; Ty_2}{\Gamma \vdash t^i{=}Ty_1 \;\delta\; t^i{=}Ty_2}\;(\textsc{exp}\mathrm{B3}) \qquad \frac{\Gamma \vdash \mathcal{M}_1 \;\delta\; \mathcal{M}_2}{\Gamma \vdash m^i{:}\mathcal{M}_1\;(ns) \;\delta\; m^i{:}\mathcal{M}_2\;(ns)}\;(\textsc{exp}\mathrm{B4})$$

$$\boxed{\Gamma \vdash Ty \;\delta\; Ty}$$

$$\frac{}{\Gamma \vdash \mathbf{int} \;\delta\; \mathbf{int}}\;(\textsc{exp}\mathrm{T}\textsc{y}1) \qquad \frac{}{\Gamma \vdash P_t \;\delta\; P_t}\;(\textsc{exp}\mathrm{T}\textsc{y}2)$$

$$\frac{\Gamma \vdash P_t \mapsto t^i{=}Ty_1 \qquad \Gamma \vdash Ty_1 \;\delta\; Ty_2}{\Gamma \vdash P_t \;\delta\; Ty_2}\;(\textsc{exp}\mathrm{T}\textsc{y}3)$$

$$\frac{\Gamma \vdash Ty_1 \;\delta\; Ty_3 \qquad \Gamma \vdash Ty_2 \;\delta\; Ty_4}{\Gamma \vdash Ty_1 \to Ty_2 \;\delta\; Ty_3 \to Ty_4}\;(\textsc{exp}\mathrm{T}\textsc{y}4)$$

$$\frac{\Gamma \vdash Ty_1 \;\delta\; Ty_3 \qquad \Gamma \vdash Ty_2 \;\delta\; Ty_4}{\Gamma \vdash Ty_1 + Ty_2 \;\delta\; Ty_3 + Ty_4}\;(\textsc{exp}\mathrm{T}\textsc{y}5)$$

$$\frac{\Gamma \vdash Ty_1 \;\delta\; Ty_3 \qquad \Gamma \vdash Ty_2 \;\delta\; Ty_4}{\Gamma \vdash Ty_1 \times Ty_2 \;\delta\; Ty_3 \times Ty_4}\;(\textsc{exp}\mathrm{T}\textsc{y}6)$$

$$\frac{\Gamma \vdash \mathcal{M}_1 \;\delta\; \mathcal{M}_2}{\Gamma \vdash \mathbf{unitT}\;\mathcal{M}_1\;(ns_1 \to ns_2) \;\delta\; \mathbf{unitT}\;\mathcal{M}_2\;(ns_1 \to ns_2)}\;(\textsc{exp}\mathrm{T}\textsc{y}7)$$

$$\boxed{\Gamma \vdash Ty \;\delta^{\mathrm{P}}\; Ty}$$

$$\frac{}{\Gamma \vdash Ty \;\delta^{\mathrm{P}}\, Ty} \qquad \frac{\Gamma \vdash P_t \mapsto Ty_1 \qquad \Gamma \vdash Ty_1 \;\delta^{\mathrm{P}}\; Ty_2}{\Gamma \vdash P_t \;\delta^{\mathrm{P}}\, Ty_2}$$

Figure 3.5. Type definition expansion structural subtyping rules.[2]

3.1.4 Type System Properties

The lemmas in this section states basic properties of the type system.

Lemma 3.1 (Subtyping)**.**

1. The relation $\Gamma \vdash \cdot\ \delta\ \cdot$ is reflexive and transitive.
2. The relation $\Gamma \vdash \cdot\ \delta^{\mathrm{P}}\ \cdot$ is reflexive and transitive. Furthermore, if Ty_1 is not a path and $\Gamma \vdash Ty_1\ \delta^{\mathrm{P}}\ Ty_2$, then $Ty_1 = Ty_2$.
3. The $<:$ relation is reflexive and transitive.
4. The $\Gamma \vdash \cdot <: \cdot$ relation is reflexive and transitive.

Definition 3.1. Typing context Γ is *well-formed* if $\epsilon \vdash \Gamma$.

Lemma 3.2 (Weakening)**.** Suppose that for all P such that $\Gamma \vdash P \mapsto B$, $\Gamma' \vdash P \mapsto B$.

1. $\Gamma; \mathcal{M}_1 \vdash Ds : \mathcal{M}_2$ implies $\Gamma'; \mathcal{M}_1 \vdash Ds : \mathcal{M}_2$
2. $\Gamma \vdash D : B$ implies $\Gamma' \vdash D : B$
3. $\Gamma \vdash E : Ty$ implies $\Gamma' \vdash E : Ty$.

Proof. By induction on the typing derivation. The rules inspect Γ in two ways: to look up paths, and to prevent identifier clashes. The hypothesis on Γ' covers the first case. In each instance of the second case, the term can be α-renamed to avoid conflicts. □

Lemma 3.3 (Narrowing)**.** If $\Gamma \vdash \mathcal{M}_1 <: \mathcal{M}_2$ then

1. $(\Gamma, \mathcal{M}_2); \epsilon \vdash Ds : \mathcal{M}_3$ implies $(\Gamma, \mathcal{M}_1); \epsilon \vdash Ds : \mathcal{M}_3$.
2. $\Gamma, \mathcal{M}_2 \vdash D : B$ implies $\Gamma, \mathcal{M}_1 \vdash D : B$.
3. $\Gamma, \mathcal{M}_2 \vdash E : Ty_1$ implies $\Gamma, \mathcal{M}_1 \vdash E : Ty_2$ and $\Gamma, \mathcal{M}_1 \vdash Ty_2 <: Ty_1$.

Lemma 3.4 (Well-formed types)**.** If Γ is well-formed then

1. $\Gamma; \mathcal{M}_1 \vdash Ds : \mathcal{M}_2$ implies $\Gamma \vdash \mathcal{M}_2$
2. $\Gamma \vdash D : B$ implies $\Gamma \vdash B$
3. $\Gamma \vdash E : Ty$ implies $\Gamma \vdash Ty$.

[2]The first-class modules of Harper and Lillibridge [1994] have an undecidable subtyping relation. My subtyping relation does not encounter the decidability problems that theirs does because it is less flexible. In particular, I separate type definition replacement from the structural subtyping relation, and I only allow definition expansion, not contraction. Increasing my subtyping relation's flexibility would make it undecidable, just as theirs is. A practical implementation of units would have to choose between using my weaker notion of subtyping, forgoing first-class units, or forgoing decidable type checking.

3.2 Operational Semantics

I specify the operational semantics of units as a small-step reduction relation with call-by-value evaluation (Table 3.2). Figure 3.6 contains the definition of values and evaluation contexts, and it also extends the language with the **rec** construct for creating recursive types and values. Figures 3.7 and 3.8 present the single-step reduction relation $\leadsto$. The rules for projections, function application, and case expressions are straightforward. Unit invocation places the unit's body into a module, and unit compounding concatenates the bodies of the linked units into a single unit body. Because unit linking can be recursive, the resulting unit can contain recursive value, type, and module definitions; thus, the reduction of a compound unit uses the **rec** construct. The values and evaluation contexts for programs (PV, PC), modules (MV, MC), definition sequences (SV, SC), definitions (DC, DV), expressions (V, EC), and the subexpressions of **compound** (UC, UV) accumulate evaluated modules and definitions as the program is reduced.

Since the operational semantics does not substitute definitions into expressions, the RLET rules lift **let**-bound definitions up into the nearest enclosing sequence of definitions (e.g., a module body).

The operational semantics relies on several auxiliary functions. The FLAT functions (Figure 3.9) flatten an evaluation context into the sequence of definitions that are in scope at the position of the hole. The RPATH, RERR1, and RERR2 rules look up the values of paths in the flattened version of the evaluation context. The second argument to the FLAT functions is the expression or definition used to fill the hole; it is necessary when the hole is in a **rec** construct since there the definition in the hole is in scope inside itself. The CTXT function (Figure 3.10) converts a definition sequence into a type context;

Table 3.2. Glossary for the module and unit operational semantics

Relation	Meaning
$D \in Ds$	the definition is in the sequence
$Prog \leadsto Prog$	single-step reduction
$Prog \twoheadrightarrow Prog$	multistep reduction
Function	**Meaning**
FLAT : $PC \times (D \cup E) \rightarrow Ds$	computes the bindings in scope at the hole
FLAT : $SC \times (D \cup E) \rightarrow Ds$	computes the bindings in scope at the hole
FLAT : $DC \times (D \cup E) \rightarrow Ds$	computes the bindings in scope at the hole
CTXT : $D \rightarrow B$	computes the binding that corresponds to a definition
CTXT : $Ds \rightarrow \mathcal{M}$	computes the module context for a definition sequence
CTXT : $PC \times (D \cup E) \rightarrow \Gamma$	computes the typing context in scope at the hole

Types extended for recursion:

$$\begin{array}{lcl} B & = & \textbf{rec}\ (B,\ldots,B) \mid \ldots \end{array}$$

Terms extended for recursion:

$$\begin{array}{lcl} D & = & \textbf{rec}\ (D\ldots D) \mid \ldots \end{array}$$

Values:

$$\begin{array}{lcl}
PV & = & MV\ldots MV \\
MV & = & \textbf{module}\ m^i\ \textbf{provide}\ ns\ \texttt{=}\ SV \\
 & \mid & \textbf{module}\ m^i\textbf{:}\mathcal{M}\ \textbf{provide}\ ns\ \texttt{=}\ SV \\
SV & = & \epsilon \mid DV\ SV \\
DV & = & x^i\textbf{:}Ty\texttt{=}V \mid t^i\texttt{=}Ty \mid MV \mid \textbf{rec}\ (DV\ldots DV) \\
V & = & \mathbb{Z} \mid \textbf{injl}\ V \mid \textbf{injr}\ V \mid (V,V) \mid \lambda x^i\textbf{:}Ty\textbf{.}E \\
 & \mid & \textbf{unit import}\ \mathcal{M}\ \textbf{export}\ \mathcal{M}\textbf{.}\ Ds \\
UV & = & V\textbf{:import}\ \mathcal{M}\ \textbf{export}\ \mathcal{M}
\end{array}$$

Evaluation Contexts:

$$\begin{array}{lcl}
PC & = & MV\ldots MV\ MC\ M\ldots M \\
MC & = & \textbf{module}\ m^i\ \textbf{provide}\ ns\ \texttt{=}\ SC \\
 & \mid & \textbf{module}\ m^i\textbf{:}\mathcal{M}\ \textbf{provide}\ ns\ \texttt{=}\ SC \\
SC & = & DC\ Ds \mid DV\ SC \\
DC & = & [] \mid x^i\textbf{:}Ty\texttt{=}EC \mid \textbf{invoke}\ EC\ \textbf{as}\ m^i\textbf{:}\mathcal{M} \mid MC \mid \textbf{rec}\ (DV\ldots DV\ DC\ D\ldots D) \\
EC & = & [] \mid \textbf{injl}\ EC \mid \textbf{injr}\ EC \\
 & \mid & \textbf{case}\ EC\ \textbf{of}\ E\ +\ E \mid \textbf{case}\ V\ \textbf{of}\ EC\ +\ E \mid \textbf{case}\ V\ \textbf{of}\ V\ +\ EC \\
 & \mid & (EC,\ E) \mid (V,\ EC) \mid \pi_1\ EC \mid \pi_2\ EC \mid EC\ E \mid V\ EC \mid \textbf{let}\ SC\ \textbf{in}\ E \\
 & \mid & \textbf{compound import}\ \mathcal{M}\ \textbf{export}\ \mathcal{M} \\
 & & \quad \textbf{link}\ UV\ldots UV\ UC\ U\ldots U\ \textbf{where}\ L\ldots L \\
UC & = & EC\textbf{:import}\ \mathcal{M}\ \textbf{export}\ \mathcal{M} \\
 & & \\
SC' & = & DC'\ Ds \mid DV\ SC' \\
DC' & = & [] \mid \textbf{rec}\ (DV\ldots DV\ DC'\ D\ldots D)
\end{array}$$

Figure 3.6. Values and evaluation contexts

$$\boxed{\mathit{Prog} \rightsquigarrow \mathit{Prog}}$$

$$\frac{\textsc{dom}(\mathcal{M}_2) \subseteq \textsc{dom}(\mathcal{M}_1)}{\begin{array}{c}\mathit{PC}[\textbf{invoke }(\textbf{unit import } \epsilon \textbf{ export } \mathcal{M}_1\textbf{.}\ \mathit{Ds})\textbf{ as } m^i\textbf{:}\mathcal{M}_2] \rightsquigarrow \\ \mathit{PC}[\textbf{module } m^i\textbf{:}\mathcal{M}_2 \textbf{ provide } \textsc{dom}\mathrm{N}(\mathcal{M}_2) \textbf{ = } \mathit{Ds}]\end{array}} \ (\textsc{Rinv})$$

$$\frac{\textsc{flat}(\mathit{PC}, P_{\mathrm{x}}) \vdash P_{\mathrm{x}} \mapsto \mathrm{x}^i\textbf{:}\mathit{Ty}\textbf{=}E \qquad E \in V}{\mathit{PC}[P_{\mathrm{x}}] \rightsquigarrow \mathit{PC}[E]} \ (\textsc{Rpath})$$

$$\frac{\textsc{flat}(\mathit{PC}, P_{\mathrm{x}}) \vdash P_{\mathrm{x}} \mapsto \mathrm{x}^i\textbf{:}\mathit{Ty}\textbf{=}E \qquad E \notin V}{\mathit{PC}[P_{\mathrm{x}}] \rightsquigarrow \textbf{error}} \ (\textsc{Rerr1})$$

$$\frac{\textsc{flat}(\mathit{PC}, P_{\mathrm{x}}) \vdash P_{\mathrm{x}} \mapsto \textbf{error}}{\mathit{PC}[P_{\mathrm{x}}] \rightsquigarrow \textbf{error}} \ (\textsc{Rerr2})$$

$$\mathit{PC}[\pi_1\textbf{(}V_1\textbf{,}\ V_2\textbf{)}] \rightsquigarrow \mathit{PC}[V_1]\ (\textsc{Rpj1}) \qquad \mathit{PC}[\pi_2\textbf{(}V_1\textbf{,}\ V_2\textbf{)}] \rightsquigarrow \mathit{PC}[V_2]\ (\textsc{Rpj2})$$

$$\mathit{PC}[\textbf{case injl } V_1 \textbf{ of } V_2\ +\ V_3] \rightsquigarrow \mathit{PC}[V_2\ V_1]\ (\textsc{Rcase1})$$

$$\mathit{PC}[\textbf{case injr } V_1 \textbf{ of } V_2\ +\ V_3] \rightsquigarrow \mathit{PC}[V_3\ V_1]\ (\textsc{Rcase2})$$

$$\mathit{PC}[(\lambda \mathrm{x}^i\textbf{:}\mathit{Ty}\textbf{.}E)\ V] \rightsquigarrow \mathit{PC}[E\{\mathrm{x}^i \leftarrow V\}]\ (\textsc{Rapp})$$

$$\frac{\begin{array}{c}\mathit{UVs} = (E_1\textbf{:import } \mathcal{M}_{1,3} \textbf{ export } \mathcal{M}_{1,4}) \cdots (E_n\textbf{:import } \mathcal{M}_{n,3} \textbf{ export } \mathcal{M}_{n,4}) \\ \forall j.0 < j \leq n \Rightarrow (E_j = \textbf{unit import } \mathcal{M}_{j,1} \textbf{ export } \mathcal{M}_{j,2}\textbf{.}\ \mathit{Ds}_j) \\ \forall j.0 < j \leq n \Rightarrow (\textsc{dom}(\mathcal{M}_{j,1}) \subseteq \textsc{dom}(\mathcal{M}_{j,3}) \wedge \textsc{dom}(\mathcal{M}_{j,4}) \subseteq \textsc{dom}(\mathcal{M}_{j,2})) \\ \mathit{Ds}' = \mathit{Ds}_1 \cdots \mathit{Ds}_n \\ \textsc{distinct}\mathrm{I}(\mathcal{M}_{0,1}\textbf{,}\textsc{ctxt}(\mathit{Ds}')) \qquad \mathit{Ds}'' = \mathit{Ds}'\{\mathit{id}_1 \leftarrow \mathit{id}_2 \mid (\mathit{id}_1 \leftarrow \mathit{id}_2) \in \mathit{Ls}\}\end{array}}{\begin{array}{c}\mathit{PC}[\textbf{compound import } \mathcal{M}_{0,1} \textbf{ export } \mathcal{M}_{0,2} \textbf{ link } \mathit{UVs} \textbf{ where } \mathit{Ls}] \rightsquigarrow \\ \mathit{PC}[\textbf{unit import } \mathcal{M}_{0,1} \textbf{ export } \mathcal{M}_{0,2}\textbf{. rec } \mathit{Ds}'']\end{array}} \ (\textsc{Rcpd})$$

Figure 3.7. One-step reduction

$$\frac{\begin{array}{c} PC = PC_2[\textbf{module}\ m^i\ \textbf{provide}\ ns\text{=}SC'] \\ \textsc{dom}\mathrm{N}(SV) \cap \textsc{dom}\mathrm{N}(SC'[x^i\textbf{:}Ty\text{=}E]) = \emptyset \end{array}}{PC[x^i\textbf{:}Ty\text{=}(\textbf{let}\ SV\ \textbf{in}\ E)] \rightsquigarrow PC[SV\ (x^i\textbf{:}Ty\text{=}E)]}\ (\textsc{Rlet1})$$

$$\frac{\begin{array}{c} PC = PC_2[\textbf{module}\ m^i\textbf{:}\mathcal{M}\ \textbf{provide}\ ns\text{=}SC'] \\ \textsc{dom}(SV) \cap \textsc{dom}(SC'[x^i\textbf{:}Ty\text{=}E]) = \emptyset \end{array}}{PC[x^i\textbf{:}Ty\text{=}(\textbf{let}\ SV\ \textbf{in}\ E)] \rightsquigarrow PC[SV\ (x^i\textbf{:}Ty\text{=}E)]}\ (\textsc{Rlet2})$$

$$\frac{PC = PC_2[\textbf{let}\ SC'\ \textbf{in}\ E_2] \qquad \textsc{dom}(SV) \cap \textsc{dom}(SC'[x^i\textbf{:}Ty\text{=}E]) = \emptyset}{PC[x^i\textbf{:}Ty\text{=}(\textbf{let}\ SV\ \textbf{in}\ E)] \rightsquigarrow PC[SV\ (x^i\textbf{:}Ty\text{=}E)]}\ (\textsc{Rlet3})$$

$$\frac{\begin{array}{c} PC = PC_2[\textbf{module}\ m^i\ \textbf{provide}\ ns\text{=}SC'] \\ \textsc{dom}\mathrm{N}(SV) \cap \textsc{dom}\mathrm{N}(SC'[\textbf{invoke}\ E\ \textbf{as}\ m^i\textbf{:}\mathcal{M}]) = \emptyset \\ \textsc{dom}(SV) \cap \textsc{dom}(\mathcal{M}) = \emptyset \end{array}}{PC[\textbf{invoke}\ (\textbf{let}\ SV\ \textbf{in}\ E)\ \textbf{as}\ m^i\textbf{:}\mathcal{M}] \rightsquigarrow PC[SV\ (\textbf{invoke}\ E\ \textbf{as}\ m^i\textbf{:}\mathcal{M})]}\ (\textsc{Rlet4})$$

$$\frac{\begin{array}{c} PC = PC_2[\textbf{module}\ m^i\textbf{:}\mathcal{M}\ \textbf{provide}\ ns\text{=}SC'] \\ \textsc{dom}(SV) \cap \textsc{dom}(SC'[\textbf{invoke}\ E\ \textbf{as}\ m^i\textbf{:}\mathcal{M}]) = \emptyset \qquad \textsc{dom}(SV) \cap \textsc{dom}(\mathcal{M}) = \emptyset \end{array}}{PC[\textbf{invoke}\ (\textbf{let}\ SV\ \textbf{in}\ E)\ \textbf{as}\ m^i\textbf{:}\mathcal{M}] \rightsquigarrow PC[SV\ (\textbf{invoke}\ E\ \textbf{as}\ m^i\textbf{:}\mathcal{M})]}\ (\textsc{Rlet5})$$

$$\frac{\begin{array}{c} PC = PC_2[\textbf{let}\ SC'\ \textbf{in}\ E_2] \\ \textsc{dom}(SV) \cap \textsc{dom}(SC'[\textbf{invoke}\ E\ \textbf{as}\ m^i\textbf{:}\mathcal{M}]) = \emptyset \qquad \textsc{dom}(SV) \cap \textsc{dom}(\mathcal{M}) = \emptyset \end{array}}{PC[\textbf{invoke}\ (\textbf{let}\ SV\ \textbf{in}\ E)\ \textbf{as}\ m^i\textbf{:}\mathcal{M}] \rightsquigarrow PC[SV\ (\textbf{invoke}\ E\ \textbf{as}\ m^i\textbf{:}\mathcal{M})]}\ (\textsc{Rlet6})$$

Figure 3.8. One-step reduction for **let**

$$PC[\textbf{injl}\ (\textbf{let}\ SV\ \textbf{in}\ E)] \rightsquigarrow PC[\textbf{let}\ SV\ \textbf{in}\ \textbf{injl}\ E]\ (\textsc{Rlet}7)$$

$$PC[\textbf{injr}\ (\textbf{let}\ SV\ \textbf{in}\ E)] \rightsquigarrow PC[\textbf{let}\ SV\ \textbf{in}\ \textbf{injr}\ E]\ (\textsc{Rlet}8)$$

$$PC[\textbf{case}\ (\textbf{let}\ SV\ \textbf{in}\ E_1)\ \textbf{of}\ E_2 + E_3] \rightsquigarrow PC[\textbf{let}\ SV\ \textbf{in}\ \textbf{case}\ E_1\ \textbf{of}\ E_2 + E_3]]\ (\textsc{Rlet}9)$$

$$PC[\textbf{case}\ V\ \textbf{of}\ (\textbf{let}\ SV\ \textbf{in}\ E_1) + E_2] \rightsquigarrow PC[\textbf{let}\ SV\ \textbf{in}\ \textbf{case}\ V\ \textbf{of}\ E_1 + E_2]\ (\textsc{Rlet}10)$$

$$PC[\textbf{case}\ V_1\ \textbf{of}\ V_2 + (\textbf{let}\ SV\ \textbf{in}\ E)] \rightsquigarrow PC[\textbf{let}\ SV\ \textbf{in}\ \textbf{case}\ V_1\ \textbf{of}\ V_2 + E]\ (\textsc{Rlet}11)$$

$$PC[\textbf{(let}\ SV\ \textbf{in}\ E_1\textbf{,}\ E_2\textbf{)}] \rightsquigarrow PC[\textbf{let}\ SV\ \textbf{in}\ \textbf{(}E_1\textbf{,}\ E_2\textbf{)}]\ (\textsc{Rlet}12)$$

$$PC[\textbf{(}V\textbf{,}\ \textbf{let}\ SV\ \textbf{in}\ E\textbf{)}] \rightsquigarrow PC[\textbf{let}\ SV\ \textbf{in}\ \textbf{(}V\textbf{,}\ E\textbf{)}]\ (\textsc{Rlet}13)$$

$$PC[\pi_1\ (\textbf{let}\ SV\ \textbf{in}\ E)] \rightsquigarrow PC[\textbf{let}\ SV\ \textbf{in}\ \pi_1\ E]\ (\textsc{Rlet}14)$$

$$PC[\pi_2\ (\textbf{let}\ SV\ \textbf{in}\ E)] \rightsquigarrow PC[\textbf{let}\ SV\ \textbf{in}\ \pi_2\ E]\ (\textsc{Rlet}15)$$

$$PC[(\textbf{let}\ SV\ \textbf{in}\ E_1)\ E_2] \rightsquigarrow PC[\textbf{let}\ SV\ \textbf{in}\ (E_1\ E_2)]\ (\textsc{Rlet}16)$$

$$PC[V\ (\textbf{let}\ SV\ \textbf{in}\ E)] \rightsquigarrow PC[\textbf{let}\ SV\ \textbf{in}\ (V\ E)]\ (\textsc{Rlet}17)$$

$$\frac{\begin{array}{c}\mathcal{L} = L_1 \ldots L_o \qquad \mathcal{U} = UV_1 \ldots UV_m\ U_1 \ldots U_n \\ U_1 = (\textbf{let}\ SV\ \textbf{in}\ E)\textbf{:import}\ \mathcal{M}_3\ \textbf{export}\ \mathcal{M}_4 \\ \mathcal{U}' = UV_1 \ldots UV_m\ U_1' \ldots U_n \qquad U_1' = E\textbf{:import}\ \mathcal{M}_3\ \textbf{export}\ \mathcal{M}_4\end{array}}{\begin{array}{c}PC[\textbf{compound import}\ \mathcal{M}_1\ \textbf{export}\ \mathcal{M}_2\ \textbf{link}\ \mathcal{U}\ \textbf{where}\ \mathcal{L}] \rightsquigarrow \\ PC[\textbf{let}\ SV\ \textbf{in compound import}\ \mathcal{M}_1\ \textbf{export}\ \mathcal{M}_2\ \textbf{link}\ \mathcal{U}'\ \textbf{where}\ \mathcal{L}]\end{array}}\ (\textsc{Rlet}18)$$

Figure 3.8. continued

$$\boxed{\textsc{flat} : \mathit{PC} \times (D \cup E) \rightarrow \mathit{DS}}$$

$$\textsc{flat}(\mathit{MV}_1 \ldots \mathit{MV}_m\ \mathit{MC}\ M_1 \ldots M_n, \mathit{redex}) = \mathit{MV}_1 \ldots \mathit{MV}_m\ \textsc{flat}(\mathit{MC}, \mathit{redex})$$

$$\boxed{\textsc{flat} : \mathit{SC} \times (D \cup E) \rightarrow \mathit{DS}}$$

$$\begin{aligned}
\textsc{flat}([], \mathit{redex}) &= \epsilon \\
\textsc{flat}(\mathit{DC}\ \mathit{Ds}, \mathit{redex}) &= \textsc{flat}(\mathit{DC}, \mathit{redex}) \\
\textsc{flat}(\mathit{DV}\ \mathit{SC}, \mathit{redex}) &= \mathit{DV}\ \textsc{flat}(\mathit{SC}, \mathit{redex})
\end{aligned}$$

$$\boxed{\textsc{flat} : \mathit{DC} \times (D \cup E) \rightarrow \mathit{DS}}$$

$$\begin{aligned}
\textsc{flat}([], \mathit{redex}) &= \epsilon \\
\textsc{flat}(x^i\mathbf{:}\mathit{Ty}\texttt{=}\mathit{EC}, \mathit{redex}) &= \textsc{flat}(\mathit{EC}, \mathit{redex}) \\
\textsc{flat}(\mathbf{invoke}\ \mathit{EC}\ \mathbf{as}\ m^i\mathbf{:}\mathcal{M}, \mathit{redex}) &= \textsc{flat}(\mathit{EC}, \mathit{redex}) \\
\textsc{flat}(\mathbf{module}\ m^i\ \mathbf{provide}\ \mathit{ns}\ \texttt{=}\ \mathit{SC}, \mathit{redex}) &= \textsc{flat}(\mathit{SC}, \mathit{redex}) \\
\textsc{flat}(\mathbf{module}\ m^i\mathbf{:}\mathcal{M}\ \mathbf{provide}\ \mathit{ns}\ \texttt{=}\ \mathit{SC}, \mathit{redex}) &= \textsc{flat}(\mathit{SC}, \mathit{redex}) \\
\textsc{flat}(\mathbf{rec}\ (\mathit{DV}_1 \ldots \mathit{DV}_m\ \mathit{DC}\ D_1\ D_n)) &= (\mathbf{rec}\ (\mathit{DV}_1 \ldots \mathit{DV}_m\ \mathit{DC}[\mathit{redex}]\ D_1\ D_n)) \\
&\ \textsc{flat}(\mathit{DC}, \mathit{redex})
\end{aligned}$$

Figure 3.9. Flattening evaluation contexts

$$\boxed{\textsc{ctxt} : D \rightarrow B}$$

$$\begin{aligned}
\textsc{ctxt}(x^i\mathbf{:}\mathit{Ty}\texttt{=}E) &= x^i\mathbf{:}\mathit{Ty} \\
\textsc{ctxt}(t^i\texttt{=}\mathit{Ty}) &= t^i\texttt{=}\mathit{Ty} \\
\textsc{ctxt}(\mathbf{invoke}\ E\ \mathbf{as}\ m^i\mathbf{:}\mathcal{M}) &= m^i\mathbf{:}\mathcal{M}\ \mathbf{provide}\ \textsc{dom}\mathrm{N}(\mathcal{M}) \\
\textsc{ctxt}(\mathbf{module}\ m^i\ \mathbf{provide}\ \mathit{ns}\ \texttt{=}\ \mathit{Ds}) &= m^i\mathbf{:}\textsc{ctxt}(\mathit{Ds})\ \mathbf{provide}\ \mathit{ns} \\
\textsc{ctxt}(\mathbf{module}\ m^i\mathbf{:}\mathcal{M}\ \mathbf{provide}\ \mathit{ns}\ \texttt{=}\ \mathit{Ds}) &= m^i\mathbf{:}\mathcal{M}\ \mathbf{provide}\ \mathit{ns} \\
\textsc{ctxt}(\mathbf{rec}\ (D_1 \ldots D_n)) &= \mathbf{rec}\ (\textsc{ctxt}(D_1)\mathbf{,} \ldots \mathbf{,}\textsc{ctxt}(D_n))
\end{aligned}$$

$$\boxed{\textsc{ctxt} : \mathit{Ds} \rightarrow \mathcal{M}}$$

$$\begin{aligned}
\textsc{ctxt}(\epsilon) &= \epsilon \\
\textsc{ctxt}(D\ \mathit{Ds}) &= \textsc{ctxt}(D)\mathbf{,}\textsc{ctxt}(\mathit{Ds})
\end{aligned}$$

$$\boxed{\textsc{ctxt} : \mathit{PC} \times (\mathit{DS} \cup E) \rightarrow \mathcal{M}}$$

$$\textsc{ctxt}(\mathit{PC}, \mathit{redex}) = \textsc{ctxt}(\textsc{flat}(\mathit{PC}, \mathit{redex}))$$

Figure 3.10. Extracting type contexts from evaluation contexts all of the necessary information is syntactically available. The definition lookup relation (Figure 3.11) looks up a path in a definition sequence. The path lookup can encounter a module defined by an **invoke** definition instead of a **module** definition; in this case the remainder of the path is unavailable, and the lookup returns **error**.

The type system of Section 3.1 does not support recursive definitions (**rec**), so I extend it in Figure 3.12 thereby allowing the results of RCMPD to be well-typed. Any well-typed program in the system of Section 3.1 is also well-typed in the extension, since the extension just adds new rules for the new construct. Thus, type soundness for the extended system implies soundness for the original system in the sense that a well-typed program will not become stuck. However, the final result of evaluation might not be well-typed in the original system, since it can contain **rec** statements introduced by RCMPD.[3] The path lookup function adds prefixes to the values it lifts out of modules, just like the type system's path lookup of Section 3.1.1. If evaluation encounters a nonvalue expression, it signals a run-time error (RERR1 and RERR2). There are many proposals to catch such ill-founded recursion errors at compile-time without restricting definitions to syntactic values [Hirschowitz et al. 2003; Boudol 2004; Dreyer 2004; Hirschowitz and Leroy 2005; Syme 2006]. Adapting these proposals to component programming with units is future work.

Taking a closer look at the RCMPD rule, it requires that the unit values' imported and exported identifiers match up with the ones specified in the import and export annotations, which themselves correspond to the identifiers used in the linkages. Furthermore, the definitions inside the units must all have distinct identifiers. These conditions can be met through α-renaming.

3.2.1 Operational Semantics Properties

The lemmas in this section state basic properties of the operational semantics.

Definition 3.2 (Redex)**.** A *redex* is an expression or definition that can appear in the hole on the left side of $\rightsquigarrow$ in Figure 3.7 or Figure 3.8.

Lemma 3.5 (Decomposition)**.**

[3]I chose the system of Section 3.1 to demonstrate that units can be type checked without special support for either value or type recursion, other than what the units provide themselves. In particular, units can be type checked without encountering any of the difficulties presented by equi-recursive types. The dynamic semantics must support evaluation of definition sequences that contain **rec** constructs, but the type checking of intermediate steps in program execution is only needed for the inductive preservation and progress proofs.

$$\boxed{Ds \vdash P \mapsto D \cup \{\mathbf{error}\}}$$

$$\frac{D \in Ds \qquad \textsc{dom}(D) = id}{Ds \vdash id \mapsto D}\ (\textsc{Dlook1})$$

$$\frac{\begin{array}{c} Ds_1 \vdash P \mapsto \mathbf{module}\ m^i\ \mathbf{provide}\ ns \texttt{=} Ds_2 \\ n \in ns \qquad D \in Ds_2 \qquad \textsc{dom}\mathrm{N}(D) = n \end{array}}{Ds_1 \vdash P.n \mapsto D\{n_2^j \leftarrow \widehat{P.n_2} \mid n_2^j \in \textsc{dom}(Ds_2)\}}\ (\textsc{Dlook2})$$

$$\frac{\begin{array}{c} Ds_1 \vdash P \mapsto \mathbf{module}\ m^i\mathbf{:}\mathcal{M}\ \mathbf{provide}\ ns \texttt{=} Ds_2 \\ n \in ns \qquad D \in Ds_2 \qquad \textsc{dom}\mathrm{N}(D) = n \qquad \textsc{dom}(D) \subseteq \textsc{dom}(\mathcal{M}) \end{array}}{Ds_1 \vdash P.n \mapsto D\{n_2^j \leftarrow \widehat{P.n_2} \mid n_2^j \in \textsc{dom}(Ds_2)\}}\ (\textsc{Dlook3})$$

$$\frac{Ds \vdash P \mapsto \mathbf{invoke}\ E\ \mathbf{as}\ m^i\mathbf{:}\mathcal{M} \qquad n \in \textsc{dom}\mathrm{N}(\mathcal{M})}{Ds \vdash P.n \mapsto \mathbf{error}}\ (\textsc{Dlook4})$$

$$\frac{Ds \vdash P \mapsto \mathbf{error}}{Ds \vdash P.n \mapsto \mathbf{error}}\ (\textsc{Dlook5})$$

Figure 3.11. Path lookup in definition sequences

$$\boxed{\Gamma \vdash B}$$

$$\frac{\textsc{distinctI}(B_1, \ldots, B_n) \qquad \forall i \leq n.\ \Gamma, \mathbf{rec}\ (B_1, \ldots, B_n) \vdash B_i}{\Gamma \vdash \mathbf{rec}\ (B_1, \ldots, B_n)}\ (\textsc{Brec})$$

$$\boxed{\Gamma; \mathcal{M} \vdash Ds : \mathcal{M}}$$

$$\frac{\begin{array}{c}\mathcal{M}_3 = \mathcal{M}_1, \textsc{ctxt}(D_1), \ldots, \textsc{ctxt}(D_n) \\ \textsc{dom}(\mathcal{M}_3) \cap \textsc{dom}(\Gamma) = \emptyset \qquad \textsc{distinctI}(\mathcal{M}_3) \\ \forall i \leq n.\ \Gamma, \mathbf{rec}\ \mathcal{M}_3 \vdash D_i : \textsc{ctxt}(D_i) \qquad (\Gamma, \mathbf{rec}\ \mathcal{M}_3); \mathcal{M}_2 \vdash Ds : \mathcal{M}_4\end{array}}{\Gamma; (\mathcal{M}_1, \mathcal{M}_2) \vdash \mathbf{rec}\ (D_1 \ldots D_n)\ Ds : (\mathbf{rec}\ \mathcal{M}_3), \mathcal{M}_4}\ (\textsc{Dsrec})$$

$$\boxed{B <: B}$$

$$\frac{\forall i \leq n.\ \exists j \leq m.\ B_j <: B'_i}{\mathbf{rec}\ (B_1, \ldots, B_m) <: \mathbf{rec}\ (B'_1, \ldots, B'_n)}\ (\textsc{sub}\text{B6})$$

$$\boxed{\Gamma \vdash B\ \delta\ B}$$

$$\frac{\forall i \leq n.\ \Gamma, \mathbf{rec}\ (B_1, \ldots, B_n) \vdash B_i\ \delta\ B'_i}{\Gamma \vdash \mathbf{rec}\ (B_1, \ldots, B_n)\ \delta\ \mathbf{rec}\ (B'_1, \ldots, B'_n)}\ (\textsc{exp}\text{Brec})$$

Figure 3.12. Type system extension

1. If $Ds \in SV$, then there exists no SC and $redex$ such that $Ds = SC[redex]$. If $Ds \notin SV$, then there exists a unique SC and $redex$ such that $Ds = SC[redex]$.
2. If $D \in DV$, then there exists no DC and $redex$ such that $D = DC[redex]$. If $D \notin DV$, then there exists a unique DC and $redex$ such that $D = DC[redex]$.
3. If $E \in V$, then there exists no EC and $redex$ such that $E = EC[redex]$. If $E \notin V$, then there exists a unique EC and $redex$ such that $E = EC[redex]$.

Proof. Statements 1–3 are proved simultaneously by induction on the structure of Ds, D, and E. □

3.3 Type Soundness

The type soundness theorem states that a well-typed will eventually reduce to a value, infinite loop, or reduce to the run-time error **error**. The proof technique uses preservation (*aka* subject reduction) and progress lemmas which combine with an induction on the number of reduction steps to yield the soundness theorem [Wright and Felleisen 1994].

Theorem 3.1 (Type Soundness)**.** If $\epsilon;\epsilon \vdash Prog_1 : \mathcal{M}$, and $Prog_1 \twoheadrightarrow Prog_2$ then either $Prog_2 = \textbf{error}$, or $\epsilon;\epsilon \vdash Prog_2 : \mathcal{M}$ and either

1. $Prog_2 \in PV$, or
2. $Prog_2 \rightsquigarrow \textbf{error}$, or
3. there exists a $Prog_3$ such that $Prog_2 \rightsquigarrow Prog_3$.

Proof. See Section 3.3.5. □

Definition 3.3. A *recursive* evaluation context has the hole under a **rec** construct. In other words, it is of the form $PC[\textbf{rec}\ (DV_1 \ldots DV_m\ []\ D_1 \ldots D_n)]$.

3.3.1 Context Function Lemmas

Lemma 3.6 (CTXT substitution)**.** If $B = \textsc{ctxt}(D)$, then $B\{m^i \leftarrow \widehat{P.m} \mid m^i \in idents\} = \textsc{ctxt}(D\{m^i \leftarrow \widehat{P.m} \mid m^i \in idents\})$.

Proof. Along with the analogous statement about Γ and Ds by structural induction on D and Ds. □

Lemma 3.7 (Type CTXT)**.**

1. If $\Gamma \vdash D : B$, then $\textsc{ctxt}(D) = B$.
2. If $\Gamma;\epsilon \vdash Ds : \mathcal{M}$, then $\textsc{ctxt}(Ds) = \mathcal{M}$.

Proof. By induction on the typing derivation. □

3.3.2 Lookup Lemmas

Lemma 3.8 (Path lookup)**.** If $\Gamma = \textsc{ctxt}(Ds)$, and $\Gamma \vdash P \mapsto B$ then either

- There exists an D such that $B = \textsc{ctxt}(D)$ and $Ds \vdash P \mapsto D$, or
- $Ds \vdash P \mapsto$ **error**.

If Γ is well-formed, the D is unique.

Proof. By rule induction on the lookup.

case $\textsc{look}1$: By structural induction on Ds.

case $\textsc{look}2$: The rules gives that $\Gamma \vdash P \mapsto m^i\mathbf{:}\mathcal{M}\ \mathbf{(}ns\mathbf{)}$. Let D' be the definition that P maps to according to the induction hypothesis. If D' is an **error** or **invoke**, the entire lookup returns **error**. Therefore, assume w.l.o.g. that D' is a module definition. By structural induction on D''s body, there is a definition D'' that corresponds to the B in $\mathcal{M}$. The final result follows by Lemma 3.6.

case $\textsc{look}3$: Same as the above case.

Uniqueness follows from the fact that a well-formed context has no duplicate identifiers in a scope, and has no unsealed module bodies with duplicate names. Although sealed module bodies can have duplicate names, their sealing description $\mathcal{M}$ cannot, and the $\textsc{Dlook}3$ rule can only return definitions whose identifier is in that $\mathcal{M}$. □

Lemma 3.9 (Well-typed lookup)**.** If $\epsilon; \epsilon \vdash Ds : \mathcal{M}$ and $Ds \vdash P \mapsto D$ then $\textsc{ctxt}(Ds) \vdash D : \textsc{ctxt}(D)$.

Proof. By rule induction on the lookup.

case $\textsc{Dlook}1$: By structural induction on Ds, using Lemma 3.2 to restore the context that follows the definition of D.

case $\textsc{Dlook}2$: In this case $P = P'\mathbf{.}n$ and $Ds \vdash P \mapsto$ **module** m^i **provide** ns=Ds'. By the induction hypothesis, the $\textsc{Dmod}1$ rule, and Lemma 3.7, $\textsc{ctxt}(Ds); \epsilon \vdash Ds' : \textsc{ctxt}(Ds')$. Since $D' \in Ds'$, $\textsc{ctxt}(Ds\ Ds') \vdash D' : \textsc{ctxt}(D')$ by the same reasoning as the above case (induction on Ds' and Lemma 3.2). To show $\textsc{ctxt}(Ds) \vdash D'\{n_2^j \leftarrow \widehat{P\mathbf{.}n_2} \mid n_2^j \in \textsc{dom}(Ds')\} : \textsc{ctxt}(D'\{n_2^j \leftarrow \widehat{P\mathbf{.}n_2} \mid n_2^j \in \textsc{dom}(Ds')\})$ induct on the derivation of D''s type and notice that any referenced identifier not found in $\textsc{ctxt}(Ds)$ must be in $\textsc{ctxt}(Ds')$, and that each of those references is replaced with a reference to Ds' through P.

case $\textsc{Dlook}3$: Same as the above case.

□

3.3.3 Progress Lemma

Lemma 3.10 (Type expansion)**.** Suppose that either $\Gamma \vdash Ty_2 <: Ty_1$ or $\Gamma \vdash Ty_1 <: Ty_2$.

1. If $\Gamma \vdash Ty_1 = \mathbf{int}$, then $\Gamma \vdash Ty_2\ \delta^{\mathrm{P}}\ \mathbf{int}$.
2. If $\Gamma \vdash Ty_1 = Ty_1' + Ty_1''$, then $\Gamma \vdash Ty_2\ \delta^{\mathrm{P}}\ Ty_2' + Ty_2''$.
3. If $\Gamma \vdash Ty_1 = Ty_1' \times Ty_1''$, then $\Gamma \vdash Ty_2\ \delta^{\mathrm{P}}\ Ty_2' \times Ty_2'',$.
4. If $\Gamma \vdash Ty_1 = Ty_1' \rightarrow Ty_1''$, then $\Gamma \vdash Ty_2\ \delta^{\mathrm{P}}\ Ty_2' \rightarrow Ty_2''$.
5. If $\Gamma \vdash Ty_1 = \mathbf{unitT}\ \mathcal{M}_1\ (ns_1 \rightarrow ns_2)$ then $\Gamma \vdash Ty_2\ \delta^{\mathrm{P}}\ \mathbf{unitT}\ \mathcal{M}_2\ (ns_3 \rightarrow ns_4)$.

Lemma 3.11 (Canonical values)**.** Suppose that $\Gamma \vdash V : Ty$.

1. $\Gamma \vdash Ty <: \mathbf{int}$ implies $V \in \mathbb{Z}$.
2. $\Gamma \vdash Ty <: Ty' \times Ty''$ implies $V = (V_1,\ V_2)$.
3. $\Gamma \vdash Ty <: Ty' + Ty''$ implies either $V = \mathbf{injl}\ V_1$, or $V = \mathbf{injr}\ V_2$.
4. $\Gamma \vdash Ty <: Ty' \rightarrow Ty''$ implies $V = \lambda x^i\mathbf{:}Ty'''\mathbf{.}E$.
5. $\Gamma \vdash Ty <: \mathbf{unitT}\ \mathcal{M}_1\ (ns_1 \rightarrow ns_2)$ implies
   $V = \mathbf{unit\ import}\ \mathcal{M}_2\ \mathbf{export}\ \mathcal{M}_3\mathbf{.}Ds$. Furthermore, $\textsc{dom}\mathrm{N}(\mathcal{M}_2) \subseteq ns_1$ and $ns_2 \subseteq \textsc{dom}\mathrm{N}(\mathcal{M}_3)$.

Lemma 3.12 (Progress)**.**

1. If $\Gamma; \epsilon \vdash Ds_1 : \mathcal{M}_1$, and $\Gamma = \textsc{ctxt}(PC, Ds_1)$, and PC is not recursive, then either $Ds_1 \in SV$, or there exists *prog* such that $PC[Ds_1] \rightsquigarrow prog$, or $PC[Ds_1] \rightsquigarrow \mathbf{error}$.
2. If $\Gamma \vdash D_1 : B$, and $\Gamma = \textsc{ctxt}(PC, D_1)$ then either $D_1 \in DV$, or there exists *prog* such that $PC[D_1] \rightsquigarrow prog$, or $PC[D_1] \rightsquigarrow \mathbf{error}$.
3. If $\Gamma \vdash E_1 : Ty$, and $\Gamma = \textsc{ctxt}(PC, E_1)$ then either $E_1 \in V$, or there exists *prog* such that $PC[E_1] \rightsquigarrow prog$, or $PC[E_1] \rightsquigarrow \mathbf{error}$.

Proof. Statements 1–3 are proved simultaneously by induction on the derivation that Ds_1, D_1, and E_1 are well-typed. (Only nonvalue cases are listed below.)

If E_1 has a nonvalue immediate subexpression E_1', then I extend the program context PC to a context, PC', that directs evaluation to the subexpression. In these cases $PC[E_1] = PC'[E_1']$, and I apply the induction hypothesis with PC' and E_1'. This requires checking that the subexpression has a type in the context $\textsc{ctxt}(PC'[E_1']) = \textsc{ctxt}(PC[E_1])\mathbf{,}\textsc{ctxt}(\ldots)$. If $PC'[E_1'] \rightsquigarrow \mathbf{error}$, then $PC[E_1] \rightsquigarrow \mathbf{error}$. Thus, $PC'[E_1'] \rightsquigarrow$ *prog* and so does the equivalent $PC[E_1]$.

case Ds1: Let $Ds_1 = D_1'\ Ds_1'$. The type rule's hypothesis ensures that, for some B_1', $\Gamma \vdash D_1' : B_1'$ and $\Gamma\mathbf{,}B_1' \vdash Ds_1' : \mathcal{M}_1'$ where $\mathcal{M}_1 = B_1'\mathbf{,}\mathcal{M}_1'$. Suppose that $D_1' \notin DV$ and let $PC' = PC[[]\ Ds_1']$. Applying the induction hypothesis according to the above discussion finished this case. If instead $D'_1 \in DV$, let $PC' = PC[D'_1\ []]$. $\textsc{ctxt}(PC', Ds'_1) = \textsc{ctxt}(PC, Ds_1)\mathbf{,}\textsc{ctxt}(D'_1) = \textsc{ctxt}(PC, Ds_1)\mathbf{,}B'_1$ (by Lemma 3.7 and the assumption that PC is not a recursive context). Again, the induction hypothesis finishes the case.

case DSREC: Let $Ds_1 = \mathbf{rec}\ (D'_1 \ldots D'_n)\ Ds''_1$. If, for all $i \leq n$, $D'_i \in DV$, the proof follows the corresponding part of the Ds1 case. Otherwise let i be the smallest number such that $D'_i \notin DV$. The type rule ensures that $\Gamma\mathbf{,rec}\ (\textsc{ctxt}(D'_1)\mathbf{,}\ldots\mathbf{,}\textsc{ctxt}(D'_n)) \vdash D'_i : \textsc{ctxt}(D'_i)$. Let $PC' = PC[\mathbf{rec}\ (D'_1 \ldots D'_{i-1}\ []\ D'_{i+1} \ldots D'_n)\ Ds''_1]$ which ensures the following.

$$\textsc{ctxt}(PC', D'_i) \quad = \quad \textsc{ctxt}(PC, Ds_1)\mathbf{,rec}\ (\textsc{ctxt}(D'_1)\mathbf{,}\ldots\mathbf{,}\textsc{ctxt}(D'_n))$$

The induction hypothesis finishes the case since this is precisely the context that D_i is checked in.

case DVAL: Let $D_1 = \mathrm{x}^i\mathbf{:}Ty\mathbf{=}E'_1$. By the rule's hypotheses, $\Gamma \vdash E'_1 : Ty'_1$. Let $PC' = PC[\mathrm{x}^i\mathbf{:}Ty\mathbf{=}[]]$.

case DINV: Let $D_1 = \mathbf{invoke}\ E'_1\ \mathbf{as}\ m^i\mathbf{:}\mathcal{M}_2$. If $E'_1 \notin V$, the conclusion follows by induction using the same argument as in the DVAL case. Now assume $E'_1 \in V$. By the rule's hypotheses, $\Gamma \vdash E'_1 : Ty'_1$ and $\Gamma \vdash Ty'_1 <: \mathbf{unitT}\ \mathcal{M}_2\ (\epsilon \rightarrow \textsc{dom}\mathrm{N}(\mathcal{M}_2))$. Lemma 3.11 ensures that E'_1 is of the form $\mathbf{unit\ import}\ \epsilon\ \mathbf{export}\ \mathcal{M}_1\mathbf{.}\ Ds$ with $\textsc{dom}\mathrm{N}(\mathcal{M}_2) \subseteq \textsc{dom}\mathrm{N}(\mathcal{M}_1)$, so the RINV rule applies as long as the side condition $\textsc{dom}(\mathcal{M}_2) \subseteq \textsc{dom}(\mathcal{M}_1)$ is met. The unit can be α-renamed to ensure that this condition holds for identifiers, since it hold for names.

case DMOD1: By the same argument as the DVAL case, using Ds in place of E.

case DMOD2: By the same argument as the DVAL case, using Ds in place of E.

case EPATH: Let $E_1 = P_{\mathrm{x}}$. The type rule ensures that, for some x^i and Ty, $\Gamma \vdash P_{\mathrm{x}} \mapsto \mathrm{x}^i\mathbf{:}Ty$. If $\textsc{flat}(PC, P_{\mathrm{x}}) \vdash P_x \mapsto \mathbf{error}$, then by reduction RERR2, $PC[P_{\mathrm{x}}] \rightsquigarrow \mathbf{error}$. Otherwise, by Lemma 3.8, there exists E' such that $\textsc{flat}(PC, P_{\mathrm{x}}) \vdash P_x \mapsto \mathrm{x}^i\mathbf{:}Ty\mathbf{=}E'$. Either rule RPATH or rule RERR1 applies.

case EPJ1: Let $E_1 = \pi_1 E'_1$. By the type rule's hypothesis $\Gamma \vdash E'_1 : Ty'_1$, and $\Gamma \vdash Ty'_1\ \delta^{\mathrm{P}}\ Ty \times Ty''_1$. If $E'_1 \in V$, Lemma 3.11 ensures that E'_1 has the form $\mathbf{(}V\mathbf{,}\ V\mathbf{)}$, and so RPJ1 applies. If $E'_1 \notin V$, let $PC' = PC[\pi_1[]]$.

case EPJ2: Similar to the EPJ1 case.

case EINJL: Similar to the EPJ1 case, except that if $E'_1 \in V$ then $E_1 \in V$ and no reduction is needed.

case EINJR: Similar to the EINJL case.

case EPROD: Let $E_1 = (E_1'\textbf{,}\ E_1'')$. By the reasoning of the EINJL case applied to E_1' if it is not a value, and applied to E_2' if E' is a value.

case ECASE: Let $E_1 =$ **case** E_1' **of** $E_1'' + E_1'''$. Similar to the EPROD case, unless E_1', E_1'', and E_1''' are all values. If they are, the type of E_1' and Lemma 3.11 ensures that either rule RCASE1 or rule RCASE2 applies.

case EAPP: Let $E_1 = E_1'\ E_1''$. Similar to EPROD if E_1' or E_1'' is not a value. If they are both values, the type of E_1' and Lemma 3.11 ensures that rule RAPP applies.

case ELET: Let $E_1 =$ **let** Ds_1' **in** $E_1'\textbf{:}Ty$. If $Ds_1' \notin SV$, then the proof follows the similar part of the DS1 case.

If $Ds_1' \in SV$ then it remains to show that one of the RLET rules applies. I proceed by case analysis on the subcontext of PC that immediately surrounds the hole (there must be one since PC cannot be equal to $[]$). For any of the EC contexts, one of the rules RLET7–RLET18 applies—except for the case of **let** $[]$ **in** E; however, placing another **let** expression directly into that hole leads to an ungrammatical program, so the possibility is ignored. The same condition applies to **module**, **rec**, and SC contexts, so only value definition and unit invocation contexts remain. In the former case one of the rules RLET1–RLET3 applies, depending on whether the sequence of definitions that the value definition is part of is inside of a module, or sealed module definition, or inside of a another **let** expression (the PC grammar admits such sequences in no other places). The precondition on domains can be met with α-renaming, noticing that for the RLET1 rule for unsigned modules, the actual name of **let** bindings might be changed. This is acceptable because they cannot be referenced from outside of the **let** expression. The case for unit invocation and RLET4–RLET6 is similar.

case ECMPD: Let $E_1 =$ **compound import** $\mathcal{M}_1$ **export** $\mathcal{M}_2$ **link** Us **where** Ls.

Let $U_1 = E'\textbf{:}\ldots$ be the first element of Us that is not in UV (supposing for now that there is one). Thus $Us = UV_1 \ldots UV_m\ U_1\ U_2 \ldots U_n$. Choose $PC' = PC[$**compound import** $\mathcal{M}_1$ **export** $\mathcal{M}_2$ **link** $UV_1 \ldots UV_m\ []\ U_2 \ldots U_n$ **where** $Ls]$. The US rule ensures that $\Gamma \vdash E' : Ty'$ so that the induction hypothesis applies.

If all of the Us are in UV then E_1 has the correct shape for rule RCMPD by Lemma 3.11 and rule US, so it remains to show that its side conditions are met. Lemma 3.11 and rule US give the necessary inclusion conditions on the unit's imported and exported names. The unit values in Us can therefore be α-renamed to ensure the corresponding inclusions on imported and exported identifiers. Furthermore, because the ECMPD rule ensures distinctness among the identifiers in the unit type annotation, and all other identifiers can freely vary, the units bodies can be α-renamed to avoid any duplicate identifiers.

$\square$

Corollary 3.1 (Whole program progress)**.** If $\epsilon; \epsilon \vdash prog_1 : \mathcal{M}$ then either $prog_1 \in PV$, $prog_1 \rightsquigarrow$ **error**, or there exists a program $prog_2$ such that $prog_1 \rightsquigarrow prog_2$.

3.3.4 Preservation Lemma

Lemma 3.13. Suppose $\Gamma \vdash Ty_2 <: Ty_1$.

1. If $\Gamma \vdash Ty_1\ \delta^{\mathrm{P}}\ Ty'_1 + Ty''_1$, then $\Gamma \vdash Ty_2\ \delta^{\mathrm{P}}\ Ty'_2 + Ty''_2$, and $\Gamma \vdash Ty'_2 <: Ty'_1$, and $\Gamma \vdash Ty''_2 <: Ty''_1$.
2. If $\Gamma \vdash Ty_1\ \delta^{\mathrm{P}}\ Ty'_1 \times Ty''_1$, then $\Gamma \vdash Ty_2\ \delta^{\mathrm{P}}\ Ty'_2 \times Ty''_2$, and $\Gamma \vdash Ty'_2 <: Ty'_1$, and $\Gamma \vdash Ty''_2 <: Ty''_1$.
3. If $\Gamma \vdash Ty_1\ \delta^{\mathrm{P}}\ Ty'_1 \rightarrow Ty''_1$, then $\Gamma \vdash Ty_2\ \delta^{\mathrm{P}}\ Ty'_2 \rightarrow Ty''_2$, and $\Gamma \vdash Ty'_1 <: Ty'_2$, and $\Gamma \vdash Ty''_2 <: Ty''_1$.

Lemma 3.14 (Unit type subtype)**.**
$\Gamma \vdash \textbf{unitT}\ \mathcal{M}_1\ (\epsilon \rightarrow \textsc{dom}\mathrm{N}(\mathcal{M}_1)) <: \textbf{unitT}\ \mathcal{M}_2\ (\epsilon \rightarrow \textsc{dom}\mathrm{N}(\mathcal{M}_2))$ implies that $\Gamma \vdash \mathcal{M}_1 <: \mathcal{M}_2$.

Lemma 3.15 (Substitution)**.** If $\Gamma, x^i{:}Ty \vdash E_1 : Ty_1$, and $\Gamma \vdash E_2 : Ty_2$, and $\Gamma \vdash Ty_2 <: Ty$, then $\Gamma \vdash E_1\{x^i \leftarrow E_2\} : Ty_3$ and $\Gamma \vdash Ty_3 <: Ty_1$.

Lemma 3.16 (Unit subtype)**.** If

1. $\Gamma \vdash$ **unit import** $\mathcal{M}_1$ **export** $\mathcal{M}_2$**.** $Ds : Ty_1$,
2. $\Gamma \vdash$ **import** $\mathcal{M}_3$ **export** $\mathcal{M}_4 : Ty_2$,
3. $\Gamma \vdash Ty_1 <: Ty_2$, and
4. $\textsc{dom}(\mathcal{M}_3) \cap \textsc{dom}(Ds) = \emptyset$

hold, then $\Gamma \vdash$ **unit import** $\mathcal{M}_3$ **export** $\mathcal{M}_4$**.** $Ds : Ty_2$.

Lemma 3.17 (Linking)**.** If

1. $\Gamma \vdash id_1 \mapsto B_1$
2. $\Gamma \vdash id_2 \mapsto B_2$
3. $\Gamma \vdash B_2 <: B_1$
4. $\Gamma \vdash D : B$

hold, then $\Gamma \vdash D\{id_1 \leftarrow id_2\} : B\{id_1 \leftarrow id_2\}$.

Lemma 3.18 (Preservation)**.**

1. $\Gamma; \epsilon \vdash Ds_1 : \mathcal{M}_1$, and PC is not recursive, and $\epsilon; \epsilon \vdash \textsc{flat}(PC, Ds_1) : \Gamma$, and $PC[Ds_1] \leadsto PC[Ds_2]$ implies $\Gamma; \epsilon \vdash Ds_2 : \mathcal{M}_2$, and $\Gamma \vdash \mathcal{M}_2 <: \mathcal{M}_1$.
2. $\Gamma \vdash D_1 : B_1$, and $\epsilon; \epsilon \vdash \textsc{flat}(PC, D_1) : \Gamma$, and $PC[D_1] \leadsto PC[D_2]$ implies $\Gamma \vdash D_2 : B_2$ and $\Gamma \vdash B_2 <: B_1$.
3. $\Gamma \vdash E_1 : Ty_1$, and $\epsilon; \epsilon \vdash \textsc{flat}(PC, E_1) : \Gamma$, and $PC[E_1] \leadsto PC[E_2]$ implies $\Gamma \vdash E_2 : Ty_2$, and $\Gamma \vdash Ty_2 <: Ty_1$.

Proof. Statements 1–3 are proved simultaneously by induction on the derivation that Ds_1, D_1, and E_1 are well-typed. In each case below, the expression (Ds_1, D_1, E_1) is either a redex, in which case a rule from Figure 3.7 or Figure 3.8 applies, or it is not, in which case one of its subexpressions is reduced. In these cases, I extend the evaluation context PC to an evaluation context PC' that wraps the subexpression. Thus, if the subexpression is E'_1, then it will be the case that $PC[E_1] = PC'[E'_1]$. Furthermore, $PC'[E'_1] \leadsto PC'[E'_2]$ for the E'_2 that gives $PC[E_2] = PC'[E'_2]$ (in other words, E_2 is just E_1 with E'_1 replaced with E'_2). I then verify that E'_1 is well-typed in the context associated with PC', and that PC' is well-formed, and apply the induction hypothesis to get the type of E'_2.

By Lemma 3.4, Γ is well-formed, and $\Gamma = \textsc{ctxt}(PC, E_1)$ (or D_1 or Ds_1 depending on the case).

Lemma 3.5 ensures that a value cannot be decomposed into a context and redex, so I omit cases for values in the following list.

case Ds1: Let $Ds_1 = D'_1\ Ds'_1$. The type rule ensures the following.

$$\begin{array}{rcl} \Gamma & \vdash & D'_1 : B'_1 \\ \Gamma, B'_1; \epsilon & \vdash & Ds'_1 : \mathcal{M}'_1 \\ \emptyset & = & \textsc{dom}(B'_1) \cap \textsc{dom}(\Gamma) \\ \mathcal{M}_1 & = & B'_1, \mathcal{M}'_1 \end{array}$$

Ds_1 cannot be a redex. Suppose that D'_1 is reduced, with $PC' = PC[[]\ Ds'_1]$. The inductive hypothesis gives $\Gamma \vdash D'_2 : B'_2$ and $\Gamma \vdash B'_2 <: B'_1$, which, along with the definition of subtyping, finishes this case.

Suppose that Ds'_1 is reduced, with $PC' = PC[D'_1\ []]$; thus, $\textsc{flat}(PC', Ds'_1) = \textsc{flat}(PC, Ds_1)\ D'_1$. Let $\Gamma' = \textsc{ctxt}(PC', Ds'_1) = \Gamma, B'_1$ (Lemma 3.7). The induction hypothesis requires that $\epsilon; \epsilon \vdash \textsc{flat}(PC', Ds'_1) : \Gamma'$, which holds by induction on the length of the definition sequence $\textsc{flat}(PC, Ds_1)$. The inductive hypothesis gives $\Gamma\textbf{,}B'_1; \epsilon \vdash Ds'_2 : \mathcal{M}'_2$ and $\Gamma \vdash \mathcal{M}'_2 <: \mathcal{M}'_1$, which, along with the definition of subtyping and Lemma 3.3 finishes this case.

case $\textsc{DsRec}$: Let $Ds_1 = \textbf{rec}\ (D'_1 \ldots D'_n)\ Ds''_1$. The type rule ensures the following.

$$\begin{array}{rcl} \mathcal{M}'_1 & = & \textsc{ctxt}(D'_1)\textbf{,}\ldots\textbf{,}\textsc{ctxt}(D_n) \\ \emptyset & = & \textsc{dom}(\Gamma) \cap \textsc{dom}(\mathcal{M}'_1) \\ & & \textsc{distinctI}(\mathcal{M}'_1) \\ \forall i \leq n.\ \Gamma\textbf{,rec}\ \mathcal{M}'_1 & \vdash & D_i : \textsc{ctxt}(D_i) \\ (\Gamma\textbf{,rec}\ \mathcal{M}'_1); \epsilon & \vdash & Ds''_1 : \mathcal{M}''_1 \\ \mathcal{M}_1 & = & (\textbf{rec}\ \mathcal{M}'_1)\textbf{,}\mathcal{M}''_1 \end{array}$$

If Ds''_1 is reduced, the proof follows the corresponding part of the $\textsc{Ds1}$ case. Otherwise, there is an i such that D'_i is reduced to D''_i. Choose PC' as follows, so that $\textsc{flat}(PC', D'_i) = \textsc{flat}(PC, Ds_1)\ \textbf{rec}\ (D'_1 \ldots D'_n)$.

$$PC' \quad = \quad PC[\textbf{rec}\ (D'_1 \ldots D'_{i-1}\ []\ D'_{i+1} \ldots D'_n)\ Ds''_1]$$

Let $\Gamma' = \textsc{ctxt}(PC', Ds'_1) = \Gamma\textbf{,rec}\ \mathcal{M}'_1$ (Lemma 3.7). The induction hypothesis requires that $\epsilon; \epsilon \vdash \textsc{flat}(PC', D'_i) : \Gamma'$, which holds by induction on n. The inductive hypothesis gives $\Gamma\textbf{,rec}\ \mathcal{M}'_1 \vdash D''_i : B'_2$ and $\Gamma\textbf{,rec}\ \mathcal{M}'_1 \vdash B'_2 <: \textsc{ctxt}(D'_i)$. Thus, the definition of subtyping and Lemma 3.3 finish the case.

case $\textsc{DVal}$: Let $D_1 = \mathrm{x}^i\textbf{:}Ty_1\textbf{=}E'_1$. By the type rule, $\Gamma \vdash E'_1 : Ty'_1$, and $\Gamma \vdash Ty'_1 <: Ty_1$, and $B_1 = \mathrm{x}^i\textbf{:}Ty_1$. Suppose that D_1 is not a redex, so that E'_1 is reduced with $PC' = PC[\mathrm{x}^i\textbf{:}Ty_1\textbf{=}[]]$. By the induction hypothesis, $\Gamma \vdash E'_2 : Ty'_2$, and $\Gamma \vdash Ty'_2 <: Ty'_1$. Thus, $\Gamma \vdash Ty'_2 <: Ty_1$ (Lemma 3.1), so D_2 also has type B_1.

If D_1 is a redex, then one of $\textsc{RLet1}$–$\textsc{RLet3}$ applies, and $E'_1 = \textbf{let}\ SV\ \textbf{in}\ E''_1$. The type rules ensure the following.

$$\begin{array}{rcl} \Gamma & \vdash & Ty_1 \\ \Gamma; \epsilon & \vdash & SV : \mathcal{M} \\ \Gamma\textbf{,}\mathcal{M} & \vdash & E''_1 : Ty''_1 \\ \Gamma\textbf{,}\mathcal{M} & \vdash & Ty''_1 <: Ty'_1 \\ \Gamma & \vdash & Ty'_1 \end{array}$$

In each case, the definitions in SV are added into a sequence of definitions SC', and the types of the bindings in the resulting sequence are unchanged. The definitions in SC' before SV have a type by the assumption (nothing has changed for them). The sequence SV also has the same type by the assumptions also because it appears in the same type context, as does the actual value definition. The definitions that follow it have the same type by Lemma 3.2 which allows the addition of the definitions in SV to the typing assumptions. The domain conditions ensure that the conditions on the sequences in their entirety are met. Thus, the resulting context is a subtype of the one prior to reduction, since it has more bindings.

case DINV: Let $D_1 = \textbf{invoke } E'_1 \textbf{ as } m^i\textbf{:}\mathcal{M}'_1$, and let $ns'_1 = \text{DOMN}(\mathcal{M}'_1)$. The type rule (along with the TUNIT rule) ensures the following.

$$\begin{array}{rcl} B_1 & = & m^i\textbf{:}\mathcal{M}'_1\ (ns'_1) \\ \Gamma & \vdash & E'_1 : Ty'_1 \\ \Gamma & \vdash & Ty'_1 <: \textbf{unitT}\ \mathcal{M}'_1\ (\epsilon \rightarrow ns'_1) \\ \Gamma & \vdash & \mathcal{M}'_1 \\ & & \text{DISTINCT}(\mathcal{M}'_1) \end{array}$$

Suppose that D_1 is a redex for RINV, so that $E'_1 = \textbf{unit import } \epsilon \textbf{ export } \mathcal{M}''_1\textbf{.}Ds'_1$, and $D_2 = \textbf{module } m^i\textbf{:}\mathcal{M}'_1 \textbf{ provide } ns'_1\textbf{=}Ds'_1$. The EUNIT type rule (along with the IE rule) ensures the following, where $ns''_1 = \text{DOMN}(\mathcal{M}''_1)$.

$$\begin{array}{rcl} Ty'_1 & = & \textbf{unitT}\ \mathcal{M}''_1\ (\epsilon \rightarrow ns''_1) \\ \Gamma & \vdash & \mathcal{M}''_1 \\ & & \text{DISTINCT}(\mathcal{M}''_1) \\ \Gamma; \epsilon & \vdash & Ds'_1 : \mathcal{M}_2 \\ \Gamma & \vdash & \mathcal{M}_2 <: \mathcal{M}''_1 \end{array}$$

The DMOD2 rule requires the following additional property for D_2 to have type B_1: $\Gamma \vdash \mathcal{M}_2 <: \mathcal{M}'_1$. Lemmas 3.14 and 3.1 ensure this.

The other possible rules that D_1 can be a redex for are RLET4–RLET6. The proof here follows the DVAL case; the extra domain condition ensures that $\mathcal{M}'_1$ remains well-formed in the scope of the additional definitions.

If D_1 is not a redex, the proof follows the DVAL case.

case DMOD1: Let D_1 = **module** m^i **provide** ns=Ds'_1. The type rule ensures the following.

$$\begin{aligned} \Gamma; \epsilon &\vdash Ds'_1 : \mathcal{M}'_1 \\ ns &\subseteq \textsc{domN}(\mathcal{M}'_1) \\ & \textsc{distinct}(\mathcal{M}'_1) \end{aligned}$$

D_1 cannot be a redex, and so Ds'_1 is reduced with $PC' = PC[$**module** m^i ... = [$]]$. The induction hypothesis ensures $\Gamma \vdash Ds'_2 : \mathcal{M}'_2$, and $\Gamma \vdash \mathcal{M}'_2 <: \mathcal{M}'_1$ which finished this case. This is sufficient as long as $\mathcal{M}'_2$ has no duplicate names. The only rules which can add new definitions into a sequence, and thereby create duplicate names are RLET1 and RLET4. Both of these rules ensure that the names that add do not duplicate any existing names.

case DMOD2: As the DMOD1 case.

case EPATH: Let $E_1 = P_x$. E_1 is a redex for (only) rule RPATH. By the EPATH rule, $\Gamma \vdash P_x \mapsto x^i{:}Ty_1$, and by the RPATH rule, $\textsc{flat}(PC, P_x) \vdash P_x \mapsto x^i{:}Ty{=}V$. By Lemma 3.8, $Ty = Ty_1$, and by Lemma 3.9 and rule DVAL $\textsc{flat}(PC, P_x) \vdash V : Ty_2$ and $\textsc{flat}(PC, P_x) \vdash Ty_2 <: Ty$.

case EPJ1: Let $E_1 = \pi_1 E'_1$. The type rule ensures that $\Gamma \vdash E'_1 : Ty'_1$ and $\Gamma \vdash Ty'_1 \; \delta^{\text{P}} \; Ty_1 \times Ty''_1$ for some Ty' and Ty''_1.

Suppose that E_1 can is a redex of the RPJ1 rule, so that $E'_1 = (V,\ V')$, and $V = E_2$. The only type rule that matches E'_1 is EPROD, and so $\Gamma \vdash E'_1 : Ty_2 \times Ty''_2$, where Ty_2 is V's type, and $Ty_2 \times Ty''_2 = Ty'_1$. Thus, $Ty_2 = Ty_1$, and so $\Gamma \vdash Ty_2 <: Ty_1$.

The other potential reduction is RLET14, in which case $E'_1 =$ **let** SV **in** E''_1 and the type rules ensure the following.

$$\begin{aligned} \Gamma; \epsilon &\vdash SV : \mathcal{M} \\ \Gamma, \mathcal{M} &\vdash E''_1 : Ty'''_1 \\ \Gamma, \mathcal{M} &\vdash Ty'''_1 <: Ty'_1 \\ \Gamma &\vdash Ty'_1 \end{aligned}$$

To finish this case, the following additional statements need to hold.

$$\begin{array}{rcl} \Gamma,\mathcal{M} & \vdash & Ty_1''' \ \delta^{\mathrm{P}} \ Ty_3 \times Ty_4 \\ \Gamma & \vdash & Ty_3 <: Ty_1 \\ \Gamma & \vdash & Ty_1 \end{array}$$

Lemmas 3.13 and 3.4 ensure that they do.

Suppose that E_1 is not a redex, and let $PC' = PC[\pi_1[]]$. By the induction hypothesis, $\Gamma \vdash E_2' : Ty_2'$, and $\Gamma \vdash Ty_2' <: Ty_1'$. By Lemma 3.13, $\Gamma \vdash Ty_2' \ \delta^{\mathrm{P}} \ Ty_2 \times Ty_2''$ and $\Gamma \vdash Ty_2 <: Ty_1$.

case EPJ2: As the EPJ1 case.

case EPROD: Let $E_1 = (E_1', E_1'')$. By the rule $\Gamma \vdash E_1' : Ty_1'$, $\Gamma \vdash E_1'' : Ty_1''$, and $Ty_1 = Ty_1' \times Ty_1''$.

Suppose that the E_1' subterm is reduced, and let $PC' = PC[([], E_1'')]$. By the induction hypothesis, $\Gamma \vdash E_2' : Ty_2'$, and $\Gamma \vdash Ty_2' <: Ty_1'$. By the EPROD rule, $Ty_2 = Ty_2' \times Ty_1''$.

The case where the E_1'' subterm is reduced is similar to the above case.

If the RLET12 rule applies, then $E_1' = \textbf{let } SV \textbf{ in } E_1'''$ and the type rules ensure the following.

$$\begin{array}{rcl} \Gamma;\epsilon & \vdash & SV : \mathcal{M} \\ \Gamma,\mathcal{M} & \vdash & E_1''' : Ty_1''' \\ \Gamma,\mathcal{M} & \vdash & Ty_1''' <: Ty_1' \\ \Gamma,\mathcal{M} & \vdash & Ty_1' \end{array}$$

To finish this case, the following additional statement needs to hold.

$$\begin{array}{rcl} \Gamma,\mathcal{M} & \vdash & Ty_1''' \times Ty_1'' <: Ty_1' \times Ty_1'' \\ \Gamma & \vdash & Ty_1 \end{array}$$

The first follows from the definition of subtyping, and the second follows from Lemma 3.4. The RLET13 case is similar.

case EINJL: Similar to the EPROD case.

case EINJR: Similar to the EPROD case.

case ECASE: Suppose E_1 is a redex of RCASE1, and let $E_1 = \textbf{case injl } V_3' \textbf{ of } V_4' + V_5'$ and $E_2 = V_4' \ V_3'$. By the ECASE and EINJL type rules, the following hold:

$$\begin{array}{rcl}\Gamma & \vdash & V_3' : Ty_3' \\ \Gamma & \vdash & V_4' : Ty_4' \\ \Gamma & \vdash & Ty_4' \ \delta^{\mathrm{P}} \ Ty_4'' \rightarrow Ty_4''' \\ \Gamma & \vdash & Ty_3' <: Ty_4'' \\ \Gamma & \vdash & Ty_4''' <: Ty_1\end{array}$$

By the EAPP rule, $\Gamma \vdash E_2 : Ty_4'''$. The case for **injr** is similar.

The RLET9–RLET11, cases are similar to the other let cases.

If E_1 is not a redex, then one of the subexpressions of the **case** expression evaluates, and the reasoning is similar to the other inductive cases.

case EAPP: Let $E_1 = E_1' \ E_1''$. The type rule ensures the following properties.

$$\begin{array}{rcl}\Gamma & \vdash & E_1' : Ty_1' \\ \Gamma & \vdash & Ty_1' \ \delta^{\mathrm{P}} \ Ty_1''' \rightarrow Ty_1 \\ \Gamma & \vdash & E_1'' : Ty_1'' \\ \Gamma & \vdash & Ty_1'' <: Ty_1'''\end{array}$$

Suppose that E_1 is a redex of RAPP, and let $E_1' = \lambda x^i{:}Ty.E_3$, so that $E_2 = E_3\{x^i \leftarrow E_1''\}$. The EFUN type rule ensures the following properties.

$$\begin{array}{rcl}\Gamma, x^i{:}Ty & \vdash & E_3 : Ty_3 \\ \Gamma & \vdash & Ty \\ x^i & \notin & \textsc{dom}(\Gamma) \\ Ty_1' & = & Ty \rightarrow Ty_3\end{array}$$

Thus, $Ty = Ty_1'''$ and $Ty_3 = Ty_1$ and Lemma 3.15 finishes the case.

If E_1 is a redex of RLET16 or RLET17, the proof follows the other let cases.

If E_1 is not a redex, let $E_1 = E_1' \ E_1''$. Suppose that E_1' is reduced and let $PC' = PC[[] \ E_1'']$. By the induction hypothesis, $\Gamma \vdash E_2' : Ty_2'$, and $\Gamma \vdash Ty_2' <: Ty_1'$. By Lemma 3.13, $Ty_2' \ \delta^{\mathrm{P}} \ Ty_2''' \rightarrow Ty_2$, and $\Gamma \vdash Ty_1''' <: Ty_2'''$, and $\Gamma \vdash Ty_2 <: Ty_1$. $\Gamma \vdash Ty_1'' <: Ty_2'''$ shows that E_2 has type Ty_2 and finishes this case.

Suppose that E'' is reduced and let $PC' = PC[E_1' \ []]$. By the induction hypothesis, $\Gamma \vdash E_2'' : Ty_2''$, and $\Gamma \vdash Ty_2'' <: Ty_1''$, which by the EAPP rule, finishes this case.

case $\textsc{Elet}$: Let $E_1 = \textbf{let } Ds'_1 \textbf{ in } E'_1$. E_1 cannot be a redex, and only Ds'_1 can be reduced. Let $PC' = PC[\textbf{let } [] \textbf{ in } E'_1]$. By induction hypothesis $\Gamma; \epsilon \vdash Ds'_2 : \mathcal{M}'_2$ and $\Gamma \vdash \mathcal{M}'_2 <: \mathcal{M}'_1$. Lemma 3.3 finishes this case.

case $\textsc{Ecmpd}$: Let $E_1 = \textbf{compound import } \mathcal{M}_{0,1} \textbf{ export } \mathcal{M}_{0,2} \textbf{ link } Us \textbf{ where } Ls$.

If E_1 is not a redex, then one of the expressions in one of the Us is reduced. By induction hypothesis, the reduced expression has a type that is a subtype of the original expression (the **compound** expression does not extend the type environment for its subexpressions). The $\textsc{Us}$ rule ensures that the original type is a subtype of the explicitly specified type, and the resulting **compound** expression satisfies this condition through subtyping transitivity.

If E_1 is a redex for $\textsc{Rlet}18$ the proof follows the other let cases.

Suppose that E_1 is a redex for $\textsc{Rcpd}$ and that $\mathcal{M}_{0,1}$, $\mathcal{M}_{0,2}$, Ds', and Ds'' are as in that rule, that $Us = UV_1 \ldots UV_n$. Expand the definitions as follows (where j ranges between 1 and n, inclusive).

$$\begin{aligned} Ds' &= D_1 \ldots D_m \\ Ds'' &= D'_1 \ldots D'_m \\ UV_j &= (\textbf{unit import } \mathcal{M}_{j,1} \textbf{ export } \mathcal{M}_{j,2}\textbf{. } Ds_j)\textbf{:import } \mathcal{M}_{j,3} \textbf{ export } \mathcal{M}_{j,4} \end{aligned}$$

Assume enough α-renaming on Ds' so that no defined identifier is also defined in the $\mathcal{M}_{j,3}$s. Thus $D'_i = D_i\{Ls\}$ where $\{Ls\}$ denotes the substitution indicated by the linkages.

The following conditions will show that the unit resulting from evaluation has the same type as the original compound, where $\mathcal{M} = \mathcal{M}_{0,1}\textbf{,}\textsc{ctxt}(D_1)\textbf{,}\ldots\textbf{,}\textsc{ctxt}(D_m)$ and $\mathcal{M}' = \mathcal{M}_{0,1}\textbf{,}\textsc{ctxt}(D'_1)\textbf{,}\ldots\textbf{,}\textsc{ctxt}(D'_m)$ and $\mathcal{M}''$ is the module description from Ty_1.

$$\begin{aligned} \Gamma &\vdash \textbf{import } \mathcal{M}_{0,1} \textbf{ export } \mathcal{M}_{0,2} : Ty_1 \\ \emptyset &= \textsc{dom}(\mathcal{M}') \cap \textsc{dom}(\Gamma) \\ & \quad\textsc{distinct}\mathrm{I}(\mathcal{M}') \\ \Gamma &\vdash (\textbf{rec } \mathcal{M}') <: \mathcal{M}'' \\ \Gamma\textbf{,rec } \mathcal{M}' &\vdash D'_i : \textsc{ctxt}(D'_i) \end{aligned}$$

The first is immediate in the $\textsc{Ecmpd}$ rule. The second by the IE type rule which ensures that the compounds type is well-formed (and hence that $\mathcal{M}_{0,1}$ does not clash with Γ) and by the EUNIT type rule which ensures that each D_i has an identifier not in Γ, since each D_i originates in a well-typed subunit. The RCPD rule ensures the third condition.

Before proceeding to the last two statements, I consider the following unit definitions.

$$E'_j \;=\; \textbf{unit import } \mathcal{M}_{j,3} \textbf{ export } \mathcal{M}_{j,4}\textbf{.}\ Ds_j$$

This E'_j unit is simply the original E_j unit, with its imports and exports replaced with the declared imports and exports from UV'_j. By Lemma 3.16 and the U rule, $\Gamma \vdash E'_j : Ty'$ where $\Gamma \vdash \textbf{import } \mathcal{M}_{j,3} \textbf{ export } \mathcal{M}_{j,4} : Ty'_j$. Now I turn to the following unit definition, which collects all of the E'_js into one unit.

$$E' \;=\; \textbf{unit import } \mathcal{M}_{1,3}\textbf{,}\ldots\textbf{,}\mathcal{M}_{n,3} \textbf{ export } \mathcal{M}_{1,4}\textbf{,}\ldots\textbf{,}\mathcal{M}_{n,4}\textbf{.}\ D_1 \ldots D_m$$

Because of potentially duplicated names, E' might not be well-typed; however, the other conditions on E' having a type do hold based on the fact that each constituent unit has a type. Thus, for some modules N_1 and N_2, the following statements are true. (The distinctness and inclusion conditions in the type and reduction rule are sufficient to avoid identifier clashes.)

$$\begin{array}{rcl} & & (\mathcal{M}_{1,3}\textbf{,}\ldots\textbf{,}\mathcal{M}_{n,3});(\mathcal{M}_{1,4}\textbf{,}\ldots\textbf{,}\mathcal{M}_{n,4})\ \textsc{interleave}\ \mathcal{N}_1 \\ \Gamma & \vdash & \mathcal{N}_1 \\ \Gamma;(\mathcal{M}_{1,3}\textbf{,}\ldots\textbf{,}\mathcal{M}_{n,3}) & \vdash & D_1 \ldots D_m : \mathcal{N}_2 \\ \Gamma & \vdash & \mathcal{N}_2 <: \mathcal{N}_1 \end{array}$$

Adding **rec** around the definitions $D_1 \ldots D_m$ does not affect the typing as **rec** only increases the number of bindings in scope and the typing rules already ensure no potentials for conflicts in the increased scope. Thus, by the DSREC rule, the following two statements hold.

$$\mathcal{N}_2 = \mathcal{M}_{1,3}\textbf{,}\ldots\textbf{,}\mathcal{M}_{n,3}\textbf{,}\textsc{ctxt}(D_1)\textbf{,}\ldots\textbf{,}\textsc{ctxt}(D_m)$$
$$\Gamma\textbf{,rec } \mathcal{N}_2 \vdash D_i : \textsc{ctxt}(D_i)$$

Furthermore, by Lemma 3.2, adding $\mathcal{M}_{0,1}$ is harmless.

$$\Gamma\textbf{,rec } (\mathcal{M}_{0,1}\textbf{,}\mathcal{N}_2) \vdash D_i : \textsc{ctxt}(D_i)$$
$$\Gamma \vdash \textbf{rec } (\mathcal{M}_{0,1}\textbf{,}\mathcal{N}_2) <: \textbf{rec } (\mathcal{M}_{0,1}\textbf{,}\mathcal{N}_1)$$

The proof proceeds in five steps.

1. Assume that no binding in the $\mathcal{M}_{j,3}$s is (or contains, in the case of modules) a translucent type definition. If one is, simply replace all references to it with its definition, then remove it along with its corresponding linkage. Any prereplacement typing or subtyping derivation holds after the replacement, because either the type path was compared using equality, or expanded first, and there can only be one expansion. The converse holds because any time the typing or subtyping derivation encounters a type path, it could have expanded out its definition. Furthermore, this process is guaranteed to succeed because the $\mathcal{M}_{j,3}$s all appear in the original program which forbids **rec** constructs.
2. $\Gamma \vdash \textbf{rec}\ (\mathcal{M}_{0,1},\mathcal{N}_2)\{Ls\} <: \textbf{rec}\ (\mathcal{M}_{0,1},\mathcal{N}_1)\{Ls\}$. The ECMPD rule ensures that the linkages substitute for identifiers that are bound to $\mathcal{M}_{j,3}$s. Let B correspond to one of these. If B is a value description, then the substitution has no effect, since value identifiers (x^i) do not appear in contexts. By the previous step, it is not a translucent type definition. If it is an opaque type definition then the subtyping derivation for $\textbf{rec}\ (\mathcal{M}_{0,1},\mathcal{N}_2)$ does not expand it out, and so some derivation will succeed without expanding the replacement identifier. If B is a module, then the above considerations apply to each of its component pieces.

   Since no $\mathcal{M}_{j,3}$ is referenced after the substitution, they can all be dropped to conclude the following.

$$\begin{aligned}\Gamma \ \vdash\ & \textbf{rec}\ (\mathcal{M}_{0,1},\textsc{ctxt}(D_1),\ldots,\textsc{ctxt}(D_m))\{Ls\} <: \\ & \textbf{rec}\ (\mathcal{M}_{0,1},\mathcal{M}_{1,4},\ldots,\mathcal{M}_{n,4})\{Ls\}\end{aligned}$$

   The substitution can be pushed in, and since no substituted value is referenced in $\mathcal{M}_{0,1}$ (a consequence of ECMPD), the following holds.

$$\begin{aligned}\Gamma \ \vdash\ & \textbf{rec}\ \mathcal{M}' <: \\ & \textbf{rec}\ (\mathcal{M}_{0,1},\textsc{ctxt}(D_1\{Ls\}),\ldots,\textsc{ctxt}(D_m\{Ls\})) <: \\ & \textbf{rec}\ (\mathcal{M}_{0,1},\mathcal{M}_{1,4}\{Ls\},\ldots,\mathcal{M}_{n,4}\{Ls\}) <: \\ & \mathcal{M}_6\end{aligned}$$

   This dispenses with the fourth overall goal of this case.

3. Let $\Gamma' = \Gamma$**,rec** $(\mathcal{M}_{0,1}$**,**$\mathcal{N}_2)\{Ls\}$. If $(id_1 \leftarrow id_2) \in Ls$ then there exist B_1 and B_2 such that $\Gamma' \vdash id_1 \mapsto B_1$ and $\Gamma' \vdash id_2 \mapsto B_2$. Furthermore, $\Gamma' \vdash B_2 <: B_1$. This follows directly from the L rule, the previous step, and Lemma 3.3.
4. $\Gamma' \vdash D_i : \textsc{ctxt}(D_i)$. Following the reasoning of step 2, any type reference replaced in the environment could not have been expanded in the derivation of Γ**,rec** $(\mathcal{M}_{0,1}$**,**$\mathcal{N}_2) \vdash D_i : \textsc{ctxt}(D_i)$, and so it does not matter what it is replaced with here.
5. Γ**,rec** $\mathcal{M}' \vdash D_i\{Ls\} : \textsc{ctxt}(D_i\{Ls\})$. Repeated application of Lemma 3.17 and steps 3 and 4 ensure that $\Gamma' \vdash D_i\{Ls\} : \textsc{ctxt}(D_i\{Ls\})$. The unreferenced $\mathcal{M}_{j,3}$ bindings in Γ' can be thrown out to yield the goal.

□

Corollary 3.2 (Whole program preservation)**.** If $\epsilon; \epsilon \vdash Prog_1 : \mathcal{M}$ and $Prog_1 \rightsquigarrow Prog_2$, then $\epsilon; \epsilon \vdash Prog_2 : \mathcal{M}$.

3.3.5 Proof of Theorem 3.1

Proof. By induction on the number of evaluation steps in the multistep reduction $\twoheadrightarrow$. If no steps are taken, then $Prog_1 = Prog_2$, and the result is immediate by Corollary 3.1. Suppose that $Prog_1 \twoheadrightarrow Prog' \rightsquigarrow Prog_2$, and that $Prog_2 \neq$ **error**. By induction, $\epsilon; \epsilon \vdash Prog' : \mathcal{M}$. By Corollary 3.2, $\epsilon; \epsilon \vdash Prog_2 : \mathcal{M}$ and the result is immediate by Corollary 3.1. □

CHAPTER 4

UNITS AND MODULES IN AN EXTENSIBLE LANGUAGE

Many of the module system requirements from the first of Chapter 2 do not mention static types; therefore any module system, including one for an untyped language, should satisfy them. The type-related criteria of Chapter 2 ensure that the types of the base language are fully integrated into the module system; in particular, the interface of a typed unit can contain types and type definitions. In general, a specification language for component interfaces necessarily parallels the programming language in which the component are implemented. When a programming language construct has a counterpart in the interface specification language, the compile-time and run-time aspects of the construct can cross component boundaries.

An extensible programming language does not have a fixed set of constructs; new features can be added to the language with external extensions to its compiler (such as with macros in Lisp and Scheme). In this setting, the interface specification language should be correspondingly extensible, allowing extension authors to maintain the correspondence between language constructs and interface constructs.

This chapter presents the design of *units* for the PLT Scheme [Flatt 2006] dialect of the Scheme programming language [Kelsey et al. 1998]. Extensions to Scheme are implemented with macros that define a new syntactic form by specifying how the form is translated into Scheme. The language of unit signatures supports macros that specify how new signature forms are translated into the core signature language.

4.1 Modules and Macros

PLT Scheme's macro system is based on the *syntax case* macro system [Dybvig et al. 1992], and its module system is designed to support separate compilation in the presence of these macros [Flatt 2002]. In this section, I describe these systems and present the design of an automatic compilation management system for modules. The compilation manager relies on the properties of the module system that prevent it from being a component system. This motivates the importance of having both modules and units. Figure 4.1 presents a simplified grammar for a subset of PLT Scheme. (In the grammar, I write x^* to indicate a sequence of 0 or more elements of x.)

4.1.1 Macros

A programmer adds a new construct to the Scheme language by defining a macro with **define-syntax**. For example, to define a two-argument, short-circuit **or** form, the programmer would write (**define-syntax** **or** *expr*) where the expression evaluates to a function that transforms (**or** $expr_1$ $expr_2$) into (**let** ((x $expr_1$)) (**if** x x $expr_2$)). When the compiler encounters an occurrence of **or**, it invokes the function, called a syntax transformer, associated with **or** and then continues to process the resulting code. The

module-def	=	(**module** *module-id module-path top-def**)
module-path	=	*string*
	|	(**lib** *string**)
top-def	=	(**require** *module-path**)
	|	(**provide** *id**)
def	=	(**define** *value-id expr*)
	|	(**define-syntax** *macro-id expr*)
	|	*expr*
	⋮	
expr	=	#f
	|	*id*
	|	(*expr expr**)
	|	(*macro-id s-expr**)
	|	(**lambda** (*value-id**) *def** *expr**)
	|	(**if** *expr expr expr*)
	|	(**let** (*b**) *def** *expr**)
	⋮	
b	=	(*value-id expr*)
s-expr	=	Scheme symbols
	|	#f
	|	(*s-expr**)
	⋮	
id, *module-id*	=	Scheme symbols
macro-id, *value-id*	=	Scheme symbols

Figure 4.1. Basic grammar for PLT Scheme's modules and macros

expression inside of a **define-syntax** definition is executed at compile time, and the identifier (e.g., **or**) is bound to the syntax transformer in the compile-time environment.

Extension definitions via **define-syntax** follow the usual lexical scoping rules of Scheme. A syntax definition can appear inside of a local scope (such as a syntax definition inside of a **lambda** or **let** expression), and it is only visible in that scope. Furthermore, the name of an extension can be shadowed in a scope with another syntax binding, or a value binding. So, if a function has an *or* parameter, it hides the **or** extension in the function's body.

To assist in the construction of syntax transformers, the **syntax-case** form supports pattern-based decomposition and construction of syntax. The **or** macro above can be expressed as follows.

```
(define-syntax or
  (lambda (stx)
    (syntax-case stx ()
      ((_ expr1 expr2)
       (syntax (let ((x expr1)) (if x x expr2)))))))
```

The (_ *expr*$_1$ *expr*$_1$) piece is a *pattern* that deconstructs the syntax *stx*, and the (**syntax** ———) piece is a *template* that constructs the result syntax.[1]

Syntax extensions can be defined recursively by having the new keyword appear in the transformer's output syntax. For example, an arbitrary-argument **or** can be defined as follows.

```
(define-syntax or
  (lambda (stx)
    (syntax-case stx ()
      ((_) (syntax #f))
      ((_ expr1 expr ...)
       (syntax  (let ((x expr1)) (if x x (or expr ...))))))))
```

In addition to being recursive, this example illustrates case dispatch in pattern matching—**or** with no arguments versus **or** with one or more arguments. The ellipses in the pattern and template indicate the preceding entity corresponds to a sequence of s-expressions.

Unlike macros in LISP or C, Scheme's macros respect lexical scoping.[2] This ensures that the author of a macro does not need to know which variables might be in scope where the macro is used, and the user of a macro does not need to know which variables

[1]I use ——— to indicate elided code because ... can occur in patterns and templates.

[2]A lexically-scoped macro system is one that is both hygienic [Kohlbecker et al. 1986] and referentially transparent [Clinger and Rees 1990].

the macro might introduce into its result. In the **or** example, the following call behaves as expected because the **if** that results from the macro is lexically-scoped at the macro's definition site.

```
(let ((if 12))
  (or (< 3 4)))
```

In a nonlexically-scoped macro system, the **if** in the expansion of **or** would refer to 12 instead of the built-in **if** conditional. The follow example produces 12 as expected because the x used at the macro's call site is bound to the lexically enclosing x instead of the x binding introduced by the macro (which refers to #f).

```
(let ((x 12))
  (or #f x))
```

To track lexical scoping information, the code consumed and produced by transformers is represented by annotated s-expressions called *syntax objects*. A transformer can directly construct syntax objects, instead of using templates, to implement a macro that intentionally violates lexical scoping. A common use is to create a binding that code at macro use site can refer to.

The **define-syntax** definition form can bind values other than syntax transformers in the compile-time environment; furthermore, a transformer can consult the value of another **define-syntax** binding during transformation. These features can be used to allow the definition of language extensions that cooperate between different macro usage sites. Pattern matching over records is a simple example of this technique.

PLT Scheme supports generative records (called structures) via the **define-struct** form [Flatt 2006] with full support for pattern matching [PLT 2006b]. This combination of records and pattern matching mirrors the corresponding features in a typical typed functional language. Each structure definition creates constructor, accessor, and predicate functions that work only with that structure. Each pattern match expression that involves the structure is compiled by the **match** macro into code that uses these functions to perform predicate checking, destructuring, and variable binding. The bindings for the structure's associated functions are relayed to the match macro with a **define-syntax** binding inserted by the structure's definition.

The following simple example, inspired by an addition expression in an interpreter, defines a structure *add-exp* with fields l and r.

```
(define-struct add-exp (l r))
———
(define (interp exp)
  (match exp
    ((struct add-exp (v1 v2))
     (+ (interp v1) (interp v2)))
    ———))
```

The **define-struct** form itself is a macro that expands, in this example, to the following.[3]

```
(define-values (make-add-exp add-exp? add-exp-l add-exp-r)
  ———)
(define-syntax add-exp
  (list (quote-syntax make-add-exp)
        (quote-syntax add-exp?)
        (list (quote-syntax add-exp-l) (quote-syntax add-exp-r))))
```

The **define-values** definition binds the structure's associated functions. The **define-syntax** definition binds the structure's name to a compile-time data structure that contains identifiers that can be used to refer to the functions. The **quote-syntax** form makes an identifier (the syntax object version of a symbol) that is bound in the scope that contains the **quote-syntax** expression. Thus, the macro that implements **match** can consult the definition of *add-exp* and use its constituent identifiers to generate the following code.

```
(define (interp exp)
  (if (add-exp? exp)
    (let ((v1 (add-exp-l exp))
          (v2 (add-exp-r exp)))
      (+ (interp v1) (interp v2)))
    ———))
```

4.1.2 Modules

Modules in PLT Scheme are semantically similar to the modules from the typed model of modules and units (Section 2.1): they are both internally-linked entities of program organization. However, there are several important syntactic differences. PLT Scheme's modules do not use the dotted notation to reference the bindings from another module; instead a module can contain a **require** statement that places all of the provided bindings of the referenced module into the requiring module's top-level scope. Furthermore, PLT Scheme's modules cannot be nested, but exist only as top-level entities.

[3]The **define-values** form supports the simultaneous definition of several variables.

Modules can provide run-time definitions created by **define** and compile-time definitions created by **define-syntax**. Thus, macros and other static information can be provided for use in other modules. For example, the above *add-exp* example can be partitioned as follows. (The second argument to the **module** form indicates the initial import for the module. In this case, *mzscheme* provides the standard PLT Scheme language.)

```
(module match mzscheme
  (provide match)
  (define-syntax match ———))

(module ast mzscheme
  (provide add-exp make-add-exp add-exp? add-exp-l add-exp-r)
  (define-struct add-exp (l r)))

(module interp mzscheme
  (require "ast.ss" (lib "match.ss"))
  (provide interp)
  (define (interp exp)
    (match exp
      ((struct add-exp (v1 v2))
       ———)
      ———)))
```

Each module is defined in a file of the same name, but with ".ss" appended to the end. The (**lib** "match.ss") requirement is to the *match* module in the standard library, and the "ast.ss" requirement is to the *ast* module in the same location as the *interp* module. Thus, the file system is used to create a global namespace which supports unambiguous references between modules.

The *ast* module must be present at the *interp* module's run time because the *interp* module uses *add-exp?* and other functions from the *ast* module. The *ast* module must also be present at *interp*'s compile time because the transformer that compiles the **match** expression relies on the value bound to *add-exp*. Hence, the compilation of a module can depend on the modules that it requires. This observation forms the basis of a compilation management strategy.

4.1.3 Compilation Management

Modules form the units of separate compilation[4] in PLT Scheme because of two properties: a module's body can only reference definitions from directly required modules,

[4] "Unit of separate compilation" is a different concept from the "unit" component system.

and the inter-module requirement relation must be acyclic. The contents of each required module, in particular their macro definitions, must be present and executable to compile the module in question. Hence, if module m_1 requires module m_2, then module m_2 can and must be compiled before m_1. Furthermore, if both are compiled, and m_2 changes, m_1 must in general be recompiled after the recompilation of m_2. This is because m_1's compilation can depend on compile-time definitions in m_2 that might have changed.

PLT Scheme's compilation manager follows the algorithm of Figure 4.2. According to this recursive algorithm, a module m will be recompiled if any module that is an ancestor in the requirement DAG needs to be recompiled; thus, compilation dependencies are transitive. The transitivity arises due to macros. In Figure 4.3, m_1 uses a macro from m_2 which is in turn defined based on a macro defined in m_3. The x in m_1 is bound to the constant 1. If m_3 changed so that the 1 was 2, then x would be bound to the constant 2, even without m_2 changing. This example can be extended to m_4, m_5, etc. with **def-def-mac**, **def-def-def-mac**, etc.

4.2 Units

Figure 4.4 presents the grammar for the basic constructs of the unit system for PLT Scheme, and Figure 4.5 presents the set example from Figure 2.2 in this syntax. Unlike the typed units of Section 2.1, the Scheme unit system is intended for practical usage; it includes signatures (created with **define-signature**) and the **rename** and **prefix** constructs for managing the names of the individual bindings imported into and exported from units. The differences between the typed and Scheme module systems imply another difference in the unit systems: a Scheme unit can only import and export bindings—not modules, and when invoked (with **define-values/invoke-unit**) a Scheme unit does not put the exported bindings into a module. It instead binds its exports in the enclosing

```
To compile module m

For each module n required by m, invoke the compilation manager for n
If one of the modules n was recompiled then
  recompile m
else if m has changed since the last compilation
  recompile m
else
  do nothing
```

Figure 4.2. Module-based compilation management

```
(module m3 mzscheme
  (provide def-mac)
  (define-syntax def-mac
    (lambda (stx)
      (syntax-case stx ()
        ((_ name)
         (syntax (define-syntax name (lambda (stx) (syntax 1)))))))))

(module m2 mzscheme
  (require "m3.ss")
  (provide mac)
  (def-mac mac))

(module m1 mzscheme
  (require "m2.ss")
  (provide x)
  (define x (mac)))
```

Figure 4.3. Modules with transitive compilation dependencies

```
      unit-expr  =  (unit (import sig-expr*) (export sig-expr*) def*)
                 |  (compound-unit (import link-spec*) (export link-id*)
                      (link linkage*))
       sig-expr  =  sig-id
                 |  (prefix symbol sig-id)
                 |  (rename sig-id rename*)
         rename  =  (id id)
        linkage  =  ((link-spec*) expr link-id*)
      link-spec  =  (link-id : sig-id)

        sig-def  =  (define-signature sig-id (spec*))
                 |  (define-signature sig-id extends sig-id (spec*))
           spec  =  value-id

            def  =  (define-values/invoke-unit expr sig-expr*)
                 |  sig-def
                 ⋮
           expr  =  (invoke-unit expr)
                 |  unit-expr
                 ⋮

link-id, sig-id  =  Scheme identifiers
```

Figure 4.4. Basic unit system grammar

```
(module order mzscheme
  (provide order-sig)
  (define-signature order-sig (compare)))

(module ordered-int mzscheme
  (require "order.ss")
  (provide oi-unit)
  (define oi-unit
    (unit (import) (export order-sig)
      (define compare ———))))

(module set mzscheme
  (require "order.ss")
  (provide set-unit set-sig)
  (define-signature set-sig (insert))
  (define set-unit
    (unit (import order-sig) (export set-sig)
      (define insert (——— compare ———)))))

(module main mzscheme
  (require "order.ss" "ordered-int.ss" "set.ss")
  (define int-set-unit
    (compound-unit (import) (export S)
      (link (((O : order-sig)) oi-unit)
            (((S : set-sig)) set-unit O))))
  (define-values/invoke-unit int-set-unit set-sig)
  (define set (insert ———)))
```

Figure 4.5. Set example

scope.

The links between units in the **compound-unit** form are not satisfied variable-by-variable, but signature-by-signature, and Scheme units take a nominal approach to unit interfaces—each signature definition creates a unique signature. In contrast, units of Chapter 2 take a structural approach. The structural approach is popular in typed, functional languages (ML's types and signatures are matched structurally), and the nominal approach is most common in object-oriented languages such as Java. Section 4.3 discusses several reasons why nominal matching is important, and Chapter 5 revisits structural matching in the Scheme system.

Each *linkage* in a compound unit's **link** clause uses *link-spec*s to specify the signatures that must be exported from that linked unit, along with *link-id*s to name them. Following the unit expression, the *link-id*s in a *linkage* specify the unit's imports; they must refer to *link-id*s defined in *link-spec*s from any of the *linkage*s or from the compound unit's **import** clause. The specified linkages in a compound unit are matched against the unit value's imports and exports by signature identity, so it does not matter which order the *link-spec*s or *link-id*s appear in.

Signatures support single-inheritence with **extends** to allow the definition of a more specific version of an interface. Much like in the typed version of units, a unit with more exports, or more specific exports (according to the declared signature extension relations), can be used in a compounding or invocation expression that expects less of it. Similarly, a unit with less imports, or less specific imports, can be used in an expression that expects more of it.

Because Scheme is untyped, error checking is split between run time and compile time. The compile-time checks are based on the signature annotations, and the run-time checks ensure that the run-time values are consistent with the annotations. Table 4.1 describes the compile-time checks performed by the unit system. The run-time checks for **compound-unit** and **define-values/invoke-unit** ensure that their subexpressions have unit values that are compatible with the specified imports and exports.

4.3 Compile-time Values in Signatures

The unit system described in Figure 4.4 and Table 4.1 supports the export of run-time value from a unit, but not the export of compile-time values. This restriction ensures that units can be independently compiled; if one of these units could export a compile-time value for a binding x, then the compilation of another unit that imports x would need to know whether x is a compile-time or run-time value (and if it is a compile-time value, what that value is). If a unit importing x was linked to two different units that export different compile-time values for x, then it would have to be compiled twice, one for each linkage.

Units in the typed language (Section 2.1) can import and export types (which are compile-time values) by placing type definitions in the unit's interface. The same idea applies to compile-time values in Scheme by allowing signatures to specify definitions for compile time values (Figure 4.6). However, there is no analogue to opaque type imports and exports because there is no use for a compile-time binding that could have *any* value.

A signature that includes definitions (both compile-time and run-time) can express more about an interface than can a signature that is simply a collection of names. For

Table 4.1. Compile-time error checking for units

define-signature

- No identifier is specified twice in a signature, including parent signatures.

unit

- Each exported binding is defined (with **define**) in the unit. This implies that no imported binding is directly exported. Such a binding would not be created by this unit, but merely passed through. If the binding was recursively linked with itself, or another unit that also did not create the binding, the binding would not exist. A definition in the unit's body, along with renaming, can avoid this problem, as in the following example.

  (**define-signature** *s* (*x*))
  (**unit** (**import** (**rename** *s* (*x* *im:x*)))
      (**export** *s*)
    (**define** *x* *im:x*))

- No binding is imported into the body more than once. Renaming and prefixing can be used to avoid this conflict when multiple imported signatures contain a specification for the same identifier.
- No imported binding is defined in the unit.
- No exported signature is the same as, or a subtype of, another exported signature. This check prevents ambiguity when compounding with the unit. Otherwise, a unit could export signatures s_1 and s_2 where s_1 is a subtype of s_2. When matching the exports to the *sig-id*s in a *link-spec*, a *sig-id* of s_2 would match both the s_1 or s_2 export.

compound-unit

- No *link-id* is specified in a *link-spec* more than once.
- Each *link-id* used in a *linkage* is specified in a *link-spec*.
- Each *link-id* used in the export clause is specified in a *linkage*'s *link-spec*.
- No *sig-id* in a *linkage* is subtype of another *sig-id* in the same *linkage*.
- No two *link-id*s on the right side of a *linkage* have associated *sig-id*s where one of the *sig-id*s is a subtype of the other. This prevents two different *link-id*s from being able to satisfy the same import.

define-values/invoke-unit

- The *sig-expr* cannot specify duplicate identifiers.

```
spec  =  value-id
      |  (define-syntax macro-id expr)
      |  (define value-id expr)
```

Figure 4.6. Definitions in signatures

example, an imported macro can codify an idiomatic usage of the imported functions. Figure 4.7, which does not use the extension of Figure 4.6, contains a simple case of such a macro taken from DrScheme's turtle graphics library [PLT 2006a]. The *splitfn* function clones the turtle, and performs its argument command on only one of the clones. The **split** macro wraps the argument command inside of a no-argument function that it passes to *splitfn*, thereby automatically implementing the necessary delayed execution of the command.[5]

The **split** macro refers to the *splitfn* function; because macros are lexically scoped, the definition of **split** must occur in the scope of the binding of *splitfn*. Hence, the macro definition is inside of *turtle-client*. This technique is unacceptable because the **split** macro must be defined inside of every unit that imports the *turtle-graphics* signature (or the unit's body will have to directly re-express the meaning of the **split** macro by directly wrapping the arguments to each call to *splitfn* with **lambda**). The macro's definition is not specific to any one implementation of the *turtle-graphics* signature and *splitfn* function. It expresses a general fact about *splitfn*, and should therefore be part of the interface of units that import and export *turtle-graphics*. Moving **split** into the signature (Figure 4.8) accomplishes this.

All of the variables mentioned in the signature are in the lexical scope of the signature's definition (along with the variables in scope around the **define-signature** definition itself). Because macros follow lexical scoping, references to plain variables (that is, those variables in a signature that are not the name of a definition) in an imported signature always refer to the value imported into the unit by that variable. In the **split** example, the *splitfn* reference in the expansion of **split** always refers to the function imported into the unit as *splitfn*. Thus, although the value can differ among linkages with various units that export the *turtle-graphics* signature, the *splitfn* binding is always to the import.

A **define-syntax** definition in a signature can define other kinds of compile-time

[5]Without delaying execution of the command, it would apply to the turtle before cloning, because of the call-by-value semantics of Scheme.

```
(define-signature graphics-primitives (———))
(define-signature turtle-graphics
  (move turn splitfn ———))

(define turtle-impl
  (unit (import graphics-primitives) (export turtle-graphics)
    (define move ———)
    (define turn ———)
    (define splitfn ———)
    ———))

(define turtle-client
  (unit (import turtle-graphics) (export)
    (define-syntax split
      (lambda (x)
        (syntax-case x ()
          ((_ args ...)
           (syntax (splitfn (lambda () args ...)))))))
     ———
     (split (move ———) (turn ———))
     ———))
```

Figure 4.7. Turtle graphics: macro in unit

```
(define-signature turtle-graphics
  (move turn splitfn ———
   (define-syntax split
     (lambda (x)
       (syntax-case x ()
         ((_ args ...)
          (syntax (splitfn (lambda () args ...)))))))))

(define turtle-client
  (unit (import turtle-graphics) (export)
    ———
    (split (move ———) (turn ———))
    ———))
```

Figure 4.8. Turtle graphics: macro in signature values besides transformers. The **define-syntax** definition in Section 4.1 that allowed the pattern matcher to deconstruct *add-exp* structures can also appear in a unit's signature. Figure 4.9 contains a sketch of an interpreter that is partitioned across several units (one for definitions, one for expressions, etc.), each of which imports the abstract syntax from a unit.[6] The **match** form inside of the units can consult the signature-based compile-time descriptions of the structures that constitute the AST.

4.3.1 Nominal Matching

The presence of definitions in unit signatures makes nominal matching semantically desirable, because it guarantees that the definitions are compatible when exported from one unit into another. With structural matching, a general compatibility check is impossible, since the contents of the definition can rely on any Scheme computation. In some cases compatibility may be unimportant, such as the *split* macro which is completely irrelevant to the exporting unit. Nominal matching also yields several other benefits [Pierce 2002] and drawbacks. I revisit the drawbacks in Chapter 5.

4.4 The Extensible Signature Language

Writing a signature for a structure by directly listing its constructor, predicate, and accessor functions along with properly encoded shape information (the **define-syntax** part) is error-prone and repetitive. Just as the **define-struct** form abstracts the pattern for creating the functions and shape information for structure creation, signatures should support a form that allows the concise description of structures. In general, a macro-based language extension can introduce its own form of compile-time information, which might need to appear in a signature. I add macro-expansion functionality to the *spec* language of signatures to allow extensions to specify how their information is expressed in signatures (Figure 4.10).

A **define-signature-form** definition associates, in the compile-time environment, its expression's value with its identifier. When that identifier is used as the first element of a *spec*, the value is applied, as a transformer, to the *spec* expression (which is a syntax object). Unlike a usual macro, the associated transformer does not return a syntax object representing a resulting definition or expression. Instead, it returns a list of syntax

[6]To motivate the implementation of the AST in a unit as opposed to a module, an alternate AST implementation might provide "smart constructors" that perform local optimization on the AST as it is constructed.

```
(define-signature ast
  (make-add-exp add-exp? add-exp-l add-exp-r
   (define-syntax add-exp
     (list (quote-syntax make-add-exp)
           (quote-syntax add-exp?)
           (list (quote-syntax add-exp-l) (quote-syntax add-exp-r))))))

(define simple-ast
  (unit (import) (export ast)
    (define-struct add-exp (l r))))

(define interp-exp
  (unit (import ast) (export ———)
    (define (interp exp)
      (match exp
        ((struct add-exp (v1 v2))
         (+ (interp v1) (interp v2)))
         ———))))

(define interp-def
  (unit (import ast) (export ———)
    ———))
```

Figure 4.9. Importing a structure

```
        spec   =   (sig-form-id s-expr*)
                   ⋮
         def   =   (define-signature-form sig-form-id expr)
                   ⋮
 sig-form-id   =   Scheme symbols
```

Figure 4.10. Macro extensible signature specifications

objects, each of which is a *spec*, to allow a signature form to specify multiple bindings. (This is similar to a macro transformer that returns several definitions and expressions by placing them into a (**begin** ———) expression.) Because the resulting *spec*s are not restricted to the basic forms (i.e., they can be applications of signature forms), a signature form can be built from other signature forms (or recursively in terms of smaller instances of itself).

In the case of structures, Figure 4.11 defines a signature form **struct** that expands to the names of the structure's functions, along with the compile-time information that describes the structure's shape. With the **struct** form, the AST's signature can be written as follows.

```
(define-signature ast
  ((struct add-exp (l r))))
```

```
(define-signature-form (struct stx)
  (syntax-case stx ()
    ((_ name (field ...))
     (begin
       (check-id (syntax name))
       (for-each check-id (syntax->list (syntax (field ...))))
       (with-syntax (((ctor-id pred-id acc-id ...) ———)
         (cons (syntax
                 (define-syntax name
                   (list (quote-syntax ctor-id)
                         (quote-syntax pred-id)
                         (list (quote-syntax acc-id) ...))))
               (syntax->list (syntax (ctor-id pred-id acc-id ...))))))))))
```

Figure 4.11. A signature form for structs

4.5 Examples

4.5.1 Views

A *view* [Wadler 1987] on a data structure is a way to manipulate it as though it were simply an algebraic data type, or a record. A view is split into two parts; the *in* part corresponds to destructuring and pattern matching, and the *out* part corresponds to construction. One simple example is to view a language's inbuilt numbers as Peano numbers built inductively from a constant *z* and a unary constructor *suc*. The **struct** form can be used to implement a unit that implements this view (Figure 4.12).

The set of ordered items from Figure 4.5 can be enriched with views (Figure 4.13). Suppose that the ordered set defines *get-min*, which returns the minimum element of the set, and *remove-min*, which returns a set that does not contain the minimum element. (In this example, the set is a purely functional (or persistent) data structure [Okasaki 1998], so that the *remove-min* operation does not alter its argument). The *min-set-sig* signature allows the set to be accessed with pattern matching as though it were a list sorted in ascending; the *max-set-sig* signature allows it to be accessed as a list sorted in descending order.

```
(module peano-numbers mzscheme
  (require (lib "unit.ss"))
  (provide z make-z suc make-suc)
  (define-signature peano-sig
    ((struct z () -setters -type)
     (struct suc (p-num) -setters -type)))
  (define peano-view
    (unit (import) (export peano-sig)
      (define (make-z) 0)
      (define (z? n) (= n 0))
      (define (make-suc n)  (+ n 1))
      (define (suc? n) (> n 0))
      (define (suc-p-num n) (− n 1))))
  (define-values/invoke-unit peano-view peano-sig))

(module example mzscheme
  (require "peano-numbers.ss" (lib "plt-match.ss"))
  (provide add)
  (define (add m n)
    (match m
      ((struct z ()) n)
      ((struct suc (msub)) (make-suc (add msub n))))))
```

Figure 4.12. Peano number view

```
(module set mzscheme
  (require "order.ss")
  (provide set-unit set-sig)
  (define-signature set-sig (insert set? get-min remove-min get-max remove-max))
  (define set-unit
    (unit (import order-sig) (export set-sig)
      (define insert (——— compare ———))
      ———)))

(module set-views mzscheme
  (require "set.ss")
  (provide min-set-sig min-set max-set-sig max-set)
  (define-signature min-set-sig
    ((struct min-set (min rest) -type -setters)))
  (define min-set
    (unit (import set) (export min-set)
      (define make-min-set insert)
      (define min-set? set?)
      (define min-set-min get-min)
      (define min-set-rest remove-min)))
  (define-signature max-set-sig
    ((struct max-set (max rest) -type -setters)))
  (define max-set ———))

(module main mzscheme
  (require "order.ss" "ordered-int.ss" "set.ss" "set-views.ss")
  (define int-set-unit
    (compound-unit (import) (export S MinS MaxS)
      (link (((O : order-sig)) oi-unit)
            (((S : set-sig)) set-unit O)
            (((MinS : min-set-sig)) min-set S)
            (((MinS : min-set-sig)) min-set S))))
  (define-values/invoke-unit int-set-unit set-sig min-set-sig max-set-sig)
  ———)
```

Figure 4.13. Views on an ordered set

The *int-set-unit* compound unit links *set-unit* with *oi-unit* as before, and it also links the export of *set-unit* (which is the primitive set interface) with the view units. Because all three set interfaces are exported from the compound unit, the user of the integer set unit can choose among, and mix, the three interfaces to the set. If the programmer who is defining *int-set-unit* did not want to offer the choice (e.g., if the *set-sig* signature exists only to afford support to multiple views and hence its interface might change), he could choose to export only the *MinS* and *MaxS* links.

4.5.2 Parameterized Language Definitions

PLT Scheme's module system supports the creation of new languages that are implemented with macros. To accomplish this, all of the constructs of a new language are defined with macros as extensions to an existing language, and a module *l* is created that exports only those macros. When *l* is used as the language argument (second argument) of the module form (instead of using *mzscheme*), the module's body can only refer to the constructs exported by *l*, which are exactly the constructs of the new language.

Occasionally, the expansions of the macros that define a new language must be parameterized over a group of functions, so that the language works differently depending on which functions are supplied. In this example, I present a language whose definition is parameterized over the implementation of its garbage collector.[7] The language definition is split into three parts: core functions for accessing memory, the garbage collection (GC) and allocation functions, and the macro that implements the language.

The language is a small subset of Scheme, and the macro-defined compilation produces Scheme with inserted calls to the GC and allocation routines, as well as to the core memory. Because the macro introduces calls to the GC and core memory, it must be defined in the scope of the GC and memory routines. A fourth entity is a program written in the language (called a mutator in GC terminology), which depends on the language macros, and hence on the GC and memory routines as well. Figure 4.14 shows the system architecture when it is implemented with modules and macros.

The *memory-core* module provides functions for dealing with the roots (locations that GC starts from), and the *allocator* module provides functions for managing the heap and allocating memory, including functions for constructing and inspecting primitive data structures (e.g., cons cells). The *gc-lang* module provides primitive constructs (**if**, **and**, **define**, etc.), it re-exports the data structure functions from the *allocator* module, and it provides primitive functions (+, −, etc.) with GC wrapping (*gc:alloc-flat* and *gc:deref*).[8] It also supplies **#%module-begin** which is a special identifier that is wrapped around the entire body of a module definition.

[7]The design and implementation of the macro- and module-based GC language are due to Greg Cooper.

[8]The allocator uses names that start with *gc:* to avoid conflicts with Scheme's built in functions. The *gc-lang* module renames them on export to use the Scheme names so that programs using the garbage collected language can refer to, for example, *cons* instead of *gc:cons*.

```
(module memory-core mzscheme
  (provide static-roots get-root-set read-root set-root! ———)
  ———)

(module allocator mzscheme
  (require "memory-core.ss")
  (provide gc:set-heap-size! gc:alloc-flat gc:deref gc:cons gc:cons? gc:first ———)
  ———)

(module gc-lang mzscheme
  (require "allocator.ss" "memory-core.ss")
  (provide if and or cond let set! define ———
           (rename gc:cons cons) (rename gc:cons? cons?) (rename gc:first first)
           (rename mem:+ +) (rename mem:- −) ———
           (rename mem:module-begin #%module-begin))
  (define (lift f) (lambda args (alloc-flat (apply f (map deref args)))))
  (define mem:+ (lift +))
  (define mem:- (lift −))
  (define-syntax mem:module-begin
    (lambda (stx)
      (define (annotate stx roots)
        (syntax-case stx (if lambda set! ———)
          ———
          ((if cnd exp . . . )
           (with-syntax (((ann-cnd (annotate (syntax cnd) roots))))
             (syntax (if (gc:deref ann-cnd) ———))))
          ———))
      (syntax-case stx (heap-size)
        ((_ (heap-size n) body . . . )
         ———
         (with-syntax (((body . . . )
                        (map (lambda (b)
                               (syntax-case b (require provide)
                                 ———
                                 (expr (annotate (——— b) '()))))
                             (syntax->list (syntax (body . . . ))))))
           (syntax (#%module-begin
                     (gc:set-heap-size! n)
                     body . . . ))))))

(module mutator "gc-lang.ss"
  (heap-size 100)
  (define (map f lst)
    (if (cons? lst)
        (cons (f (first lst)) (map f (rest lst)))
        empty)))
```

Figure 4.14. Garbage collected language: no parameterization

The *mutator* module uses `"gc-lang.ss"` as its base language (instead of *mzscheme*), which means that its entire body is wrapped with *gc-lang*'s **#%module-begin**, and that its body can only refer to functions, macros, and primitive forms exported from *gc-lang*. The **#%module-begin** expects the **heap-size** argument, and translates into a call to the allocator's *gc:set-heap-size!* function; it then calls *annotate* on the definitions and expressions in the mutator. The *annotate* function translates syntax, such as **if**, into primitive syntax with added calls to the allocator. For example, *gc-lang*'s **if** is translated into Scheme's **if** with a call to *gc:deref* placed around the **if**'s conditional expression.

In the macro and module architecture, the *gc-lang* module refers to a particular allocator implementation. The **require** statement in *gc-lang* must be altered to use *gc-lang* with a different allocator module; in particular, two different garbage collected languages cannot co-exist without copying the entire implementation of *gc-lang*.

With a unit-based architecture (Figure 4.15), a single implementation of the *gc-lang* macros can be used with different allocators. In particular, the dependency of the *gc-lang* module on the *allocator* module has been reversed. An *allocator* module implements a unit whose export signature, *gc-lang-sig*, contains the *gc-lang* module's functionality. The mutator requires the allocator, and so it can directly select between different allocation strategies without any changes to the *gc-lang* module.

The *gc-lang-sig* signature lists each function that the allocator is responsible for implementing. This places these functions in the scope of the definitions in the remainder of the signature. The *lift* function appears in the signature because it refers to *deref* and *alloc-flat* which will come from the unit that exports the signature. The primitive function definitions use *lift*, so they are also defined in the signature. Lastly, the **body** macro is exactly the same as the **mem:module-begin** macro in Figure 4.14.

The mutator from Figure 4.15 has several differences from the previous version, besides having to explicitly choose and invoke an allocator. It has to manually wrap its body with **body**, and it uses the *mzscheme* language, instead of a specially constructed one. This allows it to use any Scheme primitive bypassing the garbage collected heap.[9] The *mutator-lang* module (Figure 4.16) overcomes the problems with a custom **#%module-begin** similar to the original *gc-lang*'s.

[9]It is even worse. This example has an error from the **define-values/invoke-unit** trying to redefine module level bindings from *mzscheme*, such as *cons*.

```
(module gc-lang mzscheme
  (require "memory-core.ss")
  (provide gc-lang-sig)
  (define prim:+ +)
  (define prim:- −)
  ———
  (define-signature gc-lang-sig
    (set-heap-size! alloc-flat deref cons cons? first ———
     (define (lift f) (lambda args (alloc-flat (apply f (map deref args)))))
     (define-values (+ − ———)
       (values (lift prim:+) (lift prim:-) ———))
     (define-syntax body
       (lambda (stx)
         (define (annotate stx roots) ———)
         ———)))))

(module allocator mzscheme
  (require "memory-core.ss"
           "gc-lang.ss")
  (provide allocator)
  (define allocator
    (unit (import) (export (prefix gc: gc-lang-sig))
      (define gc:set-heap-size ———)
      ———)))

(module mutator mzscheme
  (require "allocator.ss" "gc-lang.ss")
  (define-values/invoke-unit allocator gc-lang-sig)
  (body
    (heap-size 100)
    (define (map f lst)
      (if (cons? lst)
          (cons (f (first lst)) (map f (rest lst)))
          empty))))
```

Figure 4.15. Garbage collected language: initial unit parameterization

```
(module mutator-lang mzscheme
  (provide if and or cond let set! define ———
           (rename module-begin #%module-begin))
  (define-syntax module-begin
    (lambda (stx)
      (syntax-case stx (heap-size)
        ((_ (alloc-path alloc-name) (heap-size n) x ...)
         (syntax (#%module-begin
                   (require "gc-lang.ss" alloc-path)
                   (define-values/invoke-unit alloc-name
                     (except (rename gc-lang-sig (local-body body))
                             alloc-flat lift ———))
                   (local-body
                    (heap-size n)
                    x ...))))))))

(module mutator "mutator-lang.ss"
  ("allocator.ss" allocator)
  (heap-size 100)
  (define (map f lst)
    (if (cons? lst)
        (cons (f (first lst)) (map f (rest lst)))
        empty)))
```

Figure 4.16. Garbage collected language: mutator language

A program written in the *mutator-lang* language declares which allocator to use, immediately preceding the **heap-size** declaration. The **module-begin** macro produces the code that invokes the allocation unit and places the mutator's body inside of a call to **body**. The signature for invoking the allocator renames the **body** export to **local-body**. Lexical scoping ensures that this **local-body** cannot be referenced by the mutator program. Furthermore, the **except** clause hides allocation functions that should not be visible in the mutator's body.

4.5.3 Infix Notation

Even though Scheme's syntax is fully parenthesized with prefix operators, it can be extended with a macro-defined **infix** form that recognizes infix expressions written in a typical mathematical-style notation (Figure 4.17). Following the example of SML and Haskell, new infix operators can be defined with specified associativity and precedence. Figure 4.18 contains the definition of a module *mzscheme-infix* that exports Scheme's mathematical and logical operators as infixes, along with a few other useful infix functions

```
 expr   =   (infix s-expr*)
            ⋮
  def   =   (define/infix value-id (assoc prec) expr)
        |   (define-syntax/infix macro-id (assoc prec) expr)
            ⋮
assoc   =   left
        |   right
 prec   =   real numbers
```

Figure 4.17. Infix extension

```
(module mzscheme-infix mzscheme
  (require "infix.ss")
  (provide (all-from "infix.ss")
           (all-from-except mzscheme
                            ∗ and ———)
           (rename prec-∗ ∗)
           (rename prec-and and)
           o :: ———)
  (define/infix prec-∗ (left 80)
    ∗)
  (define/infix :: (right 60) cons)
  (define/infix o (left 30)
    (lambda (f) (lambda (g) (lambda (x) (f (g x))))))
  (define-syntax/infix prec-and (left 20)
    (make-rename-transformer (syntax and)))
  ———)
```

Figure 4.18. Mzscheme infix

(function composition o, list construction $::$, etc.).[10] In the following example, x is bound to '(1 2 3).

```
(module infix-use "mzscheme-infix.ss"
  (define x (infix 1 :: 1 + 1 :: 2 ∗ 2 − 1 :: '())))
```

The **infix** macro processes the given infix expression using an operator precedence parser [Aho et al. 1986], and it returns the corresponding fully parenthesized prefix expression. The parser needs the declared precedence and associativity for each operator it encounters; because infix parsing occurs at compile time, the information must be

[10]The function *make-rename-transformer* creates a macro transformer that returns the argument identifier to *make-rename-transformer*. In *mzscheme-infix* it causes each use of **prec-and** to be replaced with **and**.

stored in the compile-time environment. The **define/infix** form uses **define-syntax** for this purpose, similar to the compile-time definition of a structure's shape in Section 4.1. For example, the definition of *::* expands to the following.

```
(define x cons)
(define-syntax ::
  (create-infix-info (quote-syntax x) 'left 80))
```

The *create-infix-info* function creates an instance of an *infix-info* structure that contains the identifier x and the precedence information for the parser. When the parser encounters an operator (e.g., *::*), it consults the compile-time environment for that binding's *infix-info* structure. If there is none, the parser assumes that the identifier is not an infix operator. The parser uses the identifier in the structure for its output syntax object (e.g., x). If an infix operator is used outside of an infix expression (e.g., (*::* 1 '())) the infix operator is directly expanded into the contained binding. This works because the *infix-info* structure has an associated procedure that executes whenever the structure is used as a function, and the procedure returns the structure's contained binding.

Infix operators can be part of a unit's interface. For example, units that implement operations for abstract algebras would specify infix precedences for the algebras' binary operations. The **sig/infix** form (Figure 4.19) specifies an infix operator along with an exported/imported function that the operator uses. The example in Figure 4.20 shows how **sig/infix** is used in the interface to a matrix unit. In this example, Scheme's rational numbers are used as the matrices' field, and $n \times n$ matrices are represented as functions from indices to field elements.

```
spec  =  (sig/infix id (assoc prec) id)
      ⋮

 (define-signature-form (sig/infix stx)
   (syntax-case stx ()
     ((_ infix-op (assoc prec) name)
      (list (syntax name)
            (syntax
             (define-syntax infix-op
               (create-infix-info (quote-syntax name) 'assoc 'prec)))))))
```

Figure 4.19. Infix declarations in signatures

```
(module abs-alg "mzscheme-infix.ss"
  (require (lib "unit.ss"))

  (define-signature F
    (zero one add-inv mul-inv
     (sig/infix ++ (left 70) add)
     (sig/infix −− (left 70) sub)
     (sig/infix ∗∗ (left 80) mul)
     (sig/infix // (left 80) div)))

  (define-signature MAT
    (dim det zero one inv
     (sig/infix m∗ (left 80) mat-mul)
     (sig/infix c∗ (left 80) con-mul)))

  (define ratF
    (unit (import) (export F)
      (define zero 0)
      (define (add x y) (+ x y))
      (define (mul x y) (∗ x y))
      ———))

(define (make-funMAT n)
  (unit (import F) (export (prefix m: MAT))
    (define m:dim n)
    (define dim-list '(1 ——— n))
    (define (m:det M) ———)
    (define m:zero (lambda (x y) zero))
    (define m:one (lambda (x y) (if (= x y) one zero)))
    (define m:inv ———)
    (define (m:mat-mul m1 m2)
      (lambda (x y)
        (apply ++
               (map
                 (lambda (j) (infix m1 x j ∗∗ m2 j y))
                 dim-list))))
    (define (m:con-mul c m)
      (lambda (x y)
        (infix c ∗∗ m x y))))))
```

Figure 4.20. A matrix unit

4.6 Contributions and Related Work

The unit system of this chapter is based on Flatt's previous unit system for PLT Scheme [Flatt and Felleisen 1998; PLT 2006b]. My system improves on the previous one by adding general support for macro definitions to signatures (which I first proposed with Culpepper and Flatt [2005]) and by supporting a general macro-expansion-based mechanism for adding new kinds of compile-time information to signatures. The compilation manager described in Section 4.1.3 is also a contribution of this chapter (the design of the separately compilable module system is due to Flatt [2002], but the design and implementation of the management algorithm for the system is novel).

Bawden [2000] proposes a system of lexically-scoped, "first-class" macros based on a type system. The "first-class" macros are statically resolvable, which preserves compilability, but the values for the bindings used in a macro's expansion can be passed into and returned from functions. A "first-class" macro is defined in a *template* that includes macro definitions and a listing of variables that the macro is parameterized over, similar to a unit's signature. Bawden's system uses types to statically track macro uses, whereas macro bindings in the unit system are immediately apparent because macro definitions are lexically embedded in signatures, which are statically attached to units.

Krishnamurthi's [2001] *unit/lang* construct allows programmers to specify the programming language of a component in addition to its external interface. A language contains new macro-like extensions and run-time primitives. A unit/lang component internally specifies which language it imports, similar to how our units specify their signatures and similar to a module's language position. However, the internally specified language position does not coordinate with the externally linked component parameters, so lexically-scoped macros cannot refer to these parameters. Our system also makes it simpler to mix together orthogonal language extensions, since there is no need to manually define a language that contains the desired combination of extensions.

Many component systems support a limited set of compile-time information in interfaces. Module and component systems in nonextensible, typed languages typically support the specification of types and algebraic datatypes in interfaces (see Chapter 2). In Java-like object-oriented languages, interfaces contain information about the methods of imported and exported classes [McDirmid et al. 2001].

The Scheme48 [Kelsey et al. 2005] module system provides some support for modules with parameterized imports. However, the signatures for the imports only contain the information that a binding is a macro, and the not macro itself. Consequently, parameterized modules that import macros cannot be independently compiled.

CHAPTER 5

A PRACTICAL UNIT SYSTEM

The unit system of Chapter 4 is designed for use in large software systems; however, in several typical situations, its use is unacceptably verbose and difficult. In this chapter, I describe three extensions that make units easier to use in these situations.

5.1 Initialization Dependencies

In Chapter 2's operational semantics of units, a run-time error could arise due to a reference to a variable whose value was not initialized. The same run-time error can occur in the Scheme unit system of Chapter 4. In general, run-time errors are entirely acceptable in untyped, safe programming languages (such as PLT Scheme), but this particular error has serious implications for component abstractions.

Figure 5.1 presents a unit definition taken from the PLT Scheme GUI application framework [Findler and Flatt 2006]. The framework is organized as a collection of 27 units; the *preferences-unit* unit implements user preference settings, and the *main-unit* unit initializes the framework, including setting default preferences. The *preferences-unit* encapsulates the state of the default values for preferences with the *default* hash table. The *main-unit* unit uses the *set-defaults* function to initialize default preferences when it is invoked. If the body of *preferences-unit* has not yet been evaluated, *defaults* will be uninitialized and result in a run-time error. The *bad-framework* unit produces this error when invoked because *main-unit* precedes *preferences-unit*; the unit's bodies are evaluated in order when invoked. The *framework* unit does not produce the error because the units are in a correct order.

In general, a unit can rely on its imports in code that executes immediately upon invocation (as in the above example) as well as in code that does not execute until the invocation is complete (for example, by calling an exported procedure whose body refers to an import). In the first case, the units that define the bindings must have their bodies initialized first. These initialization ordering constraints must be followed to successfully

```
(define-signature preferences-sig
  (set-default ———))

(define preferences-unit
  (unit (import ———) (export preferences-sig)
    (define defaults (make-hash-table))
    ———
    (define (set-defaults ———)
      ———
      (hash-table-put! defaults ———))
    ———))

(define main-unit
  (unit (import preferences-sig ———) (export ———)
    ———
    (set-defaults ———)
    ———))

(define error-framework
  (compound-unit (import ———) (export ———)
    (link ((———) main-unit P ———)
          (((P : preferences-sig) ———) preferences-unit ———)
          ———)))

(define framework
  (compound-unit (import ———) (export ———)
    (link (((P : preferences-sig) ———) preferences-unit ———)
          ((———) main-unit P ———)
          ———)))
```

Figure 5.1. Preferences for GUI applications

use the component, so they are inherently part of the unit's interface. Figure 5.2 presents the syntax of the extension to units that allows initialization dependencies to be explicitly specified as part of a unit's interface.

It is a compilation error for the **init-depend** clause to list a *sig-id* that is not imported into the unit. To avoid ambiguity, it is also a compilation error for the **import** clause to mention the same *sig-id* multiple times (even with enough prefixing and renaming to avoid imported binding conflicts).[1] Because the problem of deciding whether a given unit can use a particular import at instantiation time is undecidable (via a trivial reduction from

[1]This restriction on imports does not limit the expressiveness because the restrictions on linking prohibit linkages with duplicate signatures (Table 4.1).

$$
\begin{array}{lcl}
\textit{unit-expr} & = & (\textbf{unit}\ (\textbf{import}\ \textit{sig-expr}^*)\ (\textbf{export}\ \textit{sig-expr}^*)\ (\textbf{init-depend}\ \textit{sig-id}^*) \\
 & & \quad \textit{def}^*) \\
 & \vdots &
\end{array}
$$

Figure 5.2. Initialization dependencies

the halting problem, by constructing a unit where the first occurrence of the import in question follows the execution of a given Scheme program that may or may not halt), all initialization dependencies must be specified by the unit implementor. Explicit declaration of imports co-exists well with the idea that they are part of the unit's interface, since the dependency information must be conveyed to the potential user's of the unit. Thus, the unit implementor can both declare dependencies that cannot happen, and can omit dependencies that can happen.

When a **compound-unit** expression is executed, a run-time check ensures that each of the initialization dependencies of each of the linked unit values is satisfied by their ordering in the compound's **link** clause. A run-time error occurs if an ordering violation is detected. Because the exact unit values in the compound are not known until the compound expression is executed, it is not in general possible to move this particular class of errors from run time to compile time; however, this run-time error is a significant improvement over the uninitialized variable error that would occur without the **init-depend** clause in two ways. First, the mistake is caught when it occurs, as the erroneous compound statement is executed, instead of being detected only during a later invocation and initialization of the resulting unit. Second, the error message reports which constraint is violated, so that the programmer can easily debug the compound expression, instead of requiring the programmer to trace back the control flow that lead to the access of an uninitialized variable.

5.1.1 Static Analysis

Although my unit system requires manual specification of initialization dependencies, the design could support compile-time checking of initialization dependency specifications with a static analysis tool for Scheme programs. The key property of the analyzer is that it partitions the set of imported bindings into three categories: unused, used, and uncertain. It must not classify bindings as unused unless it is has proven that they cannot be used during initialization; a similar constraint holds for the used bindings, although it is reasonable to assume (possibly imprecisely) that each top-level statement in the unit's body is reached during initialization. A dependency on an import signature that includes only unused bindings is an error, and it is also an error to omit a dependency on a used binding. The programmer should then be informed of signatures that include only uncertain bindings; the analysis gives no insight on whether these signatures should be included or not.

5.1.2 Units and Value Recursion

The problems associated with unit initialization occur in other contexts, such as value recursion in functional programming languages (i.e., `letrec` constructs). Both a unit's body and a `letrec`'s bindings introduce recursive bindings, and a compound unit essentially concatenates several unit bodies/`letrec`s together (as per the operational semantics of Chapter 2). An unrestricted `letrec` in a call-by-value language, such as Scheme's `letrec`, experiences the same potential of encountering undefined bindings as units do.[2]

Following PLT Scheme's `letrec`,[3] the definitions in a unit's body are executed from left to right, and for a compound unit the bodies of constituent units are also executed in top-to bottom order. A well-defined, easy-to-understand ordering lets the programmer reason about any side-effects that occur during unit initialization. The drawback of a predictable execution ordering is that certain compound units (or `letrec`s) might fail, even though they could succeed with some other particular execution order, chosen on a case-by-case basis. In a `letrec` expression, the bindings can be rearranged by the programmer into the working order, just as the linked units of a compound unit can be rearranged when an initialization constraint is violated. However, the rearrangement is limited in the unit case because definitions from two different constituent units cannot be interchanged.

In the following example units, an execution order that initialized a then c then d then b would not raise an error, but this order is prohibited.

[2]Haskell's `letrec` avoids this problem altogether with lazy evaluation, and ML's `letrec` avoids it by restricting definitions to syntactic values.

[3]Standard Scheme [Kelsey et al. 1998] does not specify evaluation ordering for `letrec` bindings (or function call arguments).

```
(define-signature s (a b))
(define-signature t (c d))
(define u
  (unit (import s) (export t) (init-depend s)
    (define c 1)
    (define d a)))
(define v
  (unit (import t) (export s) (init-depend t)
    (define a 1)
    (define b c)))
(define c
  (compound-unit (import) (export)
    (link (((S : s)) u T)
          (((T : t)) v S))))
```

In fact, a system for reordering execution to avoid uninitialized variable errors amounts to lazy evaluation, which can make reasoning about side-effects difficult or impossible. Syme [2006] has studied this problem in the context of `letrec`, and shown how the programmer can direct the use of lazy evaluation in a call-by-value language to form an initialization graph that will execute in the correct order. His techniques could be applied to units by adding an optional annotation to the definitions in a unit's body that would indicate that annotated definitions are allowed to be executed out-of-order, i.e., lazily.

5.2 Link Inference

Because units are first-class values, the abstractive power of a full programming language can be applied to unit creation and linking expressions. This flexibility can obscure the exact interface of a unit value flowing into a *linkage* in a compound expression at compile time. However, units are often compounded without taking advantage of the flexibility of first-class unit values, so that a unit's definition is statically apparent from a usage site. Figure 5.3 presents an extension to the unit system that supports the definition of unit values that allow the usage site to query the unit's interface at compile time.

```
def     =  (define-unit unit-id (import sig-expr*) (export sig-expr*) def*)
        |  (define-unit-binding unit-id expr (import sig-id*) (export sig-id*))
        |  (define-compound-unit unit-id (import link-spec*) (export link-id*)
              (link linkage*))
        ⋮
unit-id =  Scheme symbols
```

Figure 5.3. First-order unit definitions

The **define-unit** and **define-compound-unit** forms create a unit value just as their **unit** and **compound-unit** counterparts, but they also bind the unit value to the specified name. Furthermore, they attach a compile-time description of the unit's interface to the name. When the name is used in a context that expects to see compile-time information, it can extract and use the information. In any other usage context the information is lost, and the name acts just as though it was created with a combination of **define** and **unit** (or **compound-unit**) instead of **define-unit** (or **define-compound-unit**).

The **define-unit-binding** forms binds an existing unit to a *unit-id*, and attaches the specified interface to it as compile-time information. Because uses of the *unit-id* might rely on the correctness of the information, **define-unit-binding** performs a run-time check that the specified interface is consistent with the given unit value (this includes checking that the expression resulted in a unit value). One possible use of **define-unit-binding** is to create a function with a unit parameter, where the parameter needs to satisfy a particular interface; constructs in the function's body can take advantage of knowledge of the interface at compile time.

```
(define (f u-param)
  (define-unit-binding u u-param (import s) (export t))
  ——— u ———)
```

The information provided by the unit definition constructs is used to infer linkages in compound units. Not only are unit dataflow patterns often simple enough to be captured with unit definitions, but linking patterns are often simple enough that particular linkages do not need to be written down at all. The **compound-unit/infer** form (Figure 5.4) allows a programmer to specify which units to link, without listing explicitly how the exports from one unit are routed into the imports of another. It uses the compile-time information attached to its constituent units to infer how they should be plugged together. For each import into one of its constituent units (including exports out of the compound unit), an export from one of the constituent units (including imports into the compound itself) with the same signature is linked into the import. If an ambiguity is present because a signature is exported from two or more of the units, the inference fails at compile time. It also fails if a needed signature export is missing.

The following example illustrates inference by giving the definition of u_4 with inference, and an equivalent unit u_5 without.

```
         unit-expr  =  (compound-unit/infer
                          (import link-spec/infer*)
                          (export export-spec/infer*)
                          (link linkage/infer*))
                    ⋮
               def  =  (define-compound-unit/infer unit-id
                          (import link-spec/infer*)
                          (export export-spec/infer*)
                          (link linkage/infer*))
                    ⋮
   link-spec/infer  =  sig-id
                    |  link-spec
 export-spec/infer  =  link-id
                    |  sig-id
     linkage/infer  =  unit-id
                    |  ((link-spec*) unit-id link-id*)
```

Figure 5.4. Link inference

```
(define-unit u1 (import q) (export r)
  ———)
(define-unit u2 (import r) (export s)
  ———)
(define-unit u3 (import q r s) (export t)
  ———)
(define-compound-unit/infer u4 (import q) (export s t)
  (link u1 u2 u3))
(define-compound-unit/infer u5 (import (lq : q)) (export ls lt)
  (link (((lr : r)) u1 lq)
        (((ls : s)) u2 lr)
        (((lt : t)) u3 lq lr ls)))
```

The ambiguities in a compound linkage can be resolved without forgoing all inference. Instead of specifying just a *unit-id* for a *linkage/infer*, the programmer can specify parts of the units interface, following the syntax of a **compound-unit**'s *linkage*. Each *link-spec* to the left of the unit specifies a *link-id* to associate with a particular exported signature. Unlike the *link-spec*s in a **compound-unit**, this listing is not exhaustive, but only serves to name a chosen subset of the unit's exported signatures. These *link-id*s can then be used on the right side of the same, or other, *unit-id*s to indicate that an import is to be satisfied with that link, the same as in the **compound-unit** case. Again, the listing is not exhaustive, so other linkages can still be inferred for other imports. In the case of ambiguity, the explicitly given linkage overrides any other linkage that could be inferred.

The framework mentioned in Section 5.1 provides an example of inference. Each of the units in the framework exports a single signature, and no two export the same signature; inference is ideal in this situation.

```
(define-compound-unit/infer framework@
  (import mred^)
  (export ———)
  (link application@ version@ color-model@ exn@ mode@ exit@ menu@
        preferences@ number-snip@ autosave@ path-utils@ icon@ keymap@
        editor@ pasteboard@ text@ color@ color-prefs@ comment-box@
        finder@ group@ canvas@ panel@ frame@ handler@ scheme@ main@))
```

Without inference the compound unit is more verbose.

```
(define-compound-unit framework@
  (import (mred : mred^))
  (export ———)
  (link
   (((application : framework:application^)) application@)
   (((version : framework:version^)) version@)
   (((color-model : framework:color-model^)) color-model@)
   (((exn : framework:exn^)) exn@)
   (((mode : framework:mode^)) mode@)
   (((exit : framework:exit^)) exit@ mred preferences)
   (((menu : framework:menu^)) menu@ mred preferences)
   (((preferences : framework:preferences^)) preferences@ mred exn exit
                                                          panel frame)
   (((number-snip : framework:number-snip^)) number-snip@ mred preferences)
   (((autosave : framework:autosave^)) autosave@ mred exit preferences frame
                                                 scheme editor text finder
                                                 group)
   (((path-utils : framework:path-utils^)) path-utils@)
   (((icon : framework:icon^)) icon@ mred)
   (((keymap : framework:keymap^)) keymap@ mred preferences finder handler
                                           frame editor)
   (((editor : framework:editor^)) editor@ mred autosave finder path-utils
                                           keymap icon preferences text
                                           pasteboard frame handler)
   (((pasteboard : framework:pasteboard^)) pasteboard@ mred editor)
   (((text : framework:text^)) text@ mred icon editor preferences keymap
                                     color-model frame scheme number-snip)
   (((color : framework:color^)) color@ preferences icon mode text
                                        color-prefs scheme)
   (((color-prefs : framework:color-prefs^)) color-prefs@ preferences editor
                                                         panel canvas)
   (((comment-box : framework:comment-box^)) comment-box@ text scheme keymap)
   (((finder : framework:finder^)) finder@ mred preferences keymap)
   (((group : framework:group^)) group@ mred application frame preferences
                                        text canvas menu)
   (((canvas : framework:canvas^)) canvas@ mred preferences frame text)
   (((panel : framework:panel^)) panel@ icon mred)
   (((frame : framework:frame^)) frame@ mred group preferences icon handler
                                           application panel finder keymap text
                                           pasteboard editor canvas menu scheme
                                           exit comment-box)
   (((handler : framework:handler^)) handler@ mred finder group text
                                              preferences frame)
   (((scheme : framework:scheme^)) scheme@ mred preferences icon keymap text
                                           editor frame comment-box mode
                                           color color-prefs)
   (((main : framework:main^)) main@ mred preferences exit group handler
                                     editor color-prefs scheme)))
```

5.2.1 Relationship with Type Systems

The unit definition functionality amounts to a type system for units that does not perform inference, but instead requires every binding to be annotated with the type of the units that can flow there. Furthermore, the type annotations are optional because units are embedded in an untyped language. As an intrinsic part of performing inference, the **compound-unit/infer** form statically checks that the compounding operation will succeed at run time. Thus, even when the linkages are fully specified in a **compound-unit/infer**, the static information about the subunits is used to ensure that the compounding operation cannot cause a run-time error. As in the above example, **define-unit-binding** can be used to give unit types to bindings, such as function parameters, that do not start with them.

5.3 Structural Signature Matching

The nominal character of signatures in unit compounding gives meaning to signatures as explicitly declared interfaces to units, and prevents unintentional linkages from working simply because they share some of the same binding names. However, the flexibility offered by structural signature matching is often convenient. For example, a unit that exports several signatures can often be treated more conveniently as a unit that exports a single signature encompassing all of the bindings of the previous signatures. Such a unit u_2 can almost be defined from u_1 using only existing techniques (Figure 5.5).

The body of u_2 first creates a unit u_3 that exports *i-sig* using the bindings imported into u_2 which are in scope throughout u_2's body, including the entirety of u_3. The u_3 unit is then linked with u_1 to form u_4 which can now be invoked, placing the definitions of y

```
(define-signature i-sig (x))
(define-signature s1 (y))
(define-signature s2 (z))
(define-signature t (y z))
(define-unit u1 (import i-sig) (export s1 s2)
  ———)

(define-unit u2 (import i-sig) (export t)
  (define-unit u3 (import) (export (rename i-sig (x i:x)))
    (define i:x x))
  (define-compound-unit/infer u4 (import) (export s1 s2)
    (link u3 u1))
  (define-values/invoke-unit u4 (prefix i: s1) (prefix i: s2))
  (define y i:y)
  (define z i:z))
```

Figure 5.5. Manual structural signature matching

and z into the body of u_2. These definitions are then used to satisfy the export signature t. This pattern takes advantage of the fact that the signatures of a noncompound unit are always matched against the body of the unit structurally to ensure the proper bindings are defined. By going though a unit's body, it can change signatures to a structurally equivalent, but nominally distinct signature.

Going a through unit's body introduces a problem, however. When u_2 is invoked, u_4 is invoked, which executes the definitions in u_3, which in turn relies on the value of the import x. Thus, u_2 has an initialization dependency on its import, unlike u_1. The **unit/new-import-export** extension (Figure 5.6) captures the desired semantics of u_2 by directly dealing comparing the signatures as a primitive form, without the extra invocation that leads to the initialization dependency.

To change a unit's import and export bindings, the **unit/new-import-export** form gives the desired import and export signatures, the unit expression to base the new unit on, and its import and export signature. A *unit/new-import-export/infer* form would be able to omit the import and export signatures for the unit expression.

```
(define-unit/new-import-export u2 (import i-sig) (export t)
  ((i-sig) u1 s1 s2))
```

To further facilitate this idiom, a signature can contain an **open** form that copies into it the contents of another signature. For example, t can be defined as follows.

```
(define-signature t ((open s1) (open s2)))
```

```
     spec  =  (open sig-expr)
              ⋮
      def  =  (provide-signature-elements sig-expr*)
           |  (define-unit/new-import-export unit-id
                (import sig-expr*)(export sig-expr*)
                ((sig-expr*) expr sig-expr*))
           |  (define-values/invoke-unit/use-context expr
                (import sig-expr*) (export sig-expr*))
           |  (define-unit-from-context unit-id sig-expr)
unit-expr  =  (unit-from-context sig-expr)
           |  (unit/new-import-export (import sig-expr*) (export sig-expr*)
                ((sig-expr*) expr sig-expr*))
              ⋮
```

Figure 5.6. Structural matching

The **open** *spec* is defined with **define-signature-form** as a signature extension similar to the **struct** form.

The framework provides a larger example of the use of structural matching. Since it is composed of many units, each exporting a different signature, the overall framework unit must export many different signatures. This gives clients of the framework unit maximum flexibility in just using part of the framework, but in most cases a client simply wants to import the entire framework. This idiom can appear whenever a unit is built from several units for purposes of internal organization, but should be provided to the external clients with a unified interface. Thus, the framework has two definitions, as follows.

```
(define-compound-unit/infer framework-separate@
  (import mred^)
  (export framework:application^
          framework:version^
          framework:color-model^
          ———
          framework:main^)
  (link ———))

(define-signature framework^
  ((open framework:application^)
   ———
   (open framework:main^)))

(define-unit/new-import-export framework@ (import mred^) (export framework^)
  ((framework:application^ ———) framework-separate@ mred^))
```

5.3.1 Convenience Forms

Figure 5.5 uses several techniques for managing import and export bindings that are useful in general. First, creating a unit that exports bindings from the context (u_3) is awkward manually. The **unit-from-context** creates a unit with the given export signature by using in-scope variables to satisfy the exports. Thus, u_3 can be defined as follows.

```
(define-unit-from-context u3 i-sig)
```

Second, invoking a unit by using the context to satisfy its imports (instead of invoking only no import units) can be accomplished with **define-values/invoke-unit/use-context**, which would support the definition of u_2 as follows (it does not resolve the initialization problem as **unit/new-import-export** did).

```
(define-unit u2 (import i-sig) (export t)
  (define-values/invoke-unit/use-context u1
    (import i-sig)
    (export (prefix i: s1) (prefix i: s2)))
  (define y i:y)
  (define z i:z))
```

The **provide-signature-elements** form is a module-level companion to **define-values/invoke-unit**. It specifies that the contents of a signature are provided from the enclosing module.

CHAPTER 6

IMPLEMENTATION AND EXPERIENCE

To explore units in an extensible language, I have implemented the unit system described in Chapters 4 and 5 as a language extension to PLT Scheme. Because PLT Scheme already supported a unit-based component system, I ported a significant portion of a major PLT Scheme application, DrScheme [Findler et al. 2002], from the previous system to my own.

The unit system is implemented using Scheme macros and functions to translate unit expressions and definitions into primitive Scheme — without any changes to the underlying interpreter or compiler. The ability to deploy the new constructs without changes to the underlying architecture is the primary advantage of the macro-based implementation. A secondary advantage is the automatic management of variable scoping provided by syntax objects; however, the translation relies on low-level syntax object manipulation features of the macro system in order to perform some manual scope management.

6.1 Units

The basic strategy for translating units and compound units into functions follows the original unit system [Flatt and Felleisen 1998]. A unit value becomes a function value, and the unit's imports and exports are stored in mutable reference cells. The unit-function creates reference cells to contain its exported values, and it returns these cells along with another function that contains the unit's body. The unit-body function takes in reference cells that contain the unit's imported values, and the body of this function places the values of exported definitions into the corresponding cells as they are executed. Because the exported cells are created before the unit's body is invoked, they can be passed into other units by a **compound-unit** expression before their associated unit body has executed.

Using reference cells for imports and exports allows units to mimic the standard **let** and **set!** implementation of **letrec** in Scheme; the cells themselves correspond to bindings that are resolved immediately upon being linked to other units. However, to support the extensible recursive scope that is crucial for unit bodies, the values in the cells are not resolved until the unit is invoked and the cells are mutated.

Figure 6.1 contains an example signature and unit definition, and it shows what they expand to. The i^1 identifier is introduced by the **define-signature** macro to hold a unique run-time identifier for this signature, and the signature's name, *se*, is bound to a structure that describes it. This includes the identifiers that hold the compile-time and run-time information (*se* and i^1), as well as the identifiers that the signature specifies for import/export (x).

```
(define-signature se (x y))
(define-signature si (a))
(define u
  (unit (import si) (export se)
    (define y (lambda () x))
    (define x (lambda () a))))
```

expands to

```
(define i^1 (gensym))
(define-syntax se
  (make-signature
    (make-siginfo (list (quote-syntax se))
                  (list (quote-syntax i^1)))
    (list (quote-syntax x) (quote-syntax y))))
―――――
(define u
  (make-unit
    (vector (vector i^2))
    (vector (vector i^1))
    (lambda ()
      (let ((exp^1 (box undefined))
            (exp^2 (box undefined)))
        (values
          (lambda (import-table)
            (let ((imp^1 (vector-ref (hash-table-get import-table i^2) 0)))
              (set-box! exp^2 (lambda () (unbox exp^1)))
              (set-box! exp^1 (lambda () (unbox imp^1)))))
          (hash-table (i^1 . (vector exp^1 exp^2))))))))
```

Figure 6.1. Unit compilation

When a unit imports or exports a particular signature, say *se*, it can query the compile-time environment for *se*'s information. It uses the list of signature names to implement the compile-time checks that prohibit certain duplicate import or export signatures. It uses the run-time bindings (e.g., i^1) in *make-unit* code it produces so that each unit value has its signature's run-time tags attached. This information enables the various run-time checks that ensure that compatible unit values flow into certain expressions (for example, as a subunit in a compound expression). The list of names (e.g., (**quote-syntax** x) (**quote-syntax** y)) allows the unit macro to compile the unit's body by redirecting uses and definitions of those names to the appropriate reference cells. In this case, a corresponds to cell imp^1, x to exp^1, and y to exp^2. This mapping is accomplished in a multistage process that is further discussed in Section 6.2.

Because a **compound-unit** expression can supply the imports in any order, the unit-body-function takes in a single value: a hash-table that maps a run-time signature identifier to a vector that contains the cells for that signature. Similarly, the export cells are stored in a hash-table of vectors to accommodate arbitrary orderings for the export-linkage bindings in **compound-unit** expressions.

Since a signature can extend another signature, a unit might receive an import whose signature extends the declared signature for that import. The cells are stored in the vector in the order declared in the signature, with the cells of the extension following those of the extended signature. Thus, the unit safely ignores the unexpected extension. The single element lists in the definition of *se* contain, in general, all of the compile-time and run-time identifiers that are associated with the signature's parents. Similarly, the innermost single element vectors in the *make-unit* call are for parent signatures (the outermost vectors have one vector entry per imported/exported signature).

6.2 Signature Macros

The signature s in Figure 6.2 contains definitions in addition to imported/exported variables. Because the u unit imports s, it can refer to the imports f, g, x, y, and **m**. The **unit** macro does not directly traverse the unit's body; instead it introduces the macro definitions in the **let-syntax** expression to have the macro system perform the traversal. Thus, immediately after expanding the **unit** macro, the body still contains references to f, g, and x. Further macro expansions replace these variables, which are bound by the **let-syntax** expression, with the corresponding *unbox* expressions (the *make-id-mapper*

```
(define-signature s
  (f g x
   (define y (f x))
   (define-syntax m
     (syntax-rules ()
       ((_ arg) (lambda () (g (+ y arg))))))))
(define u
  (unit (import s) (export)
    ——— f ——— g ——— x ——— y ———
    (m ———)
    ———))
```

expands to

———

```
(define u
  (make-unit
    (vector (vector i^1))
    (vector)
    (lambda ()
      (let ()
        (values
          (lambda (import-table)
            (let ((imp^1 (vector-ref (hash-table-get import-table i^1) 0))
                  (imp^2 (vector-ref (hash-table-get import-table i^1) 1))
                  (imp^3 (vector-ref (hash-table-get import-table i^1) 2)))
              (let-syntax ((f (make-id-mapper (quote-syntax (unbox imp^1))))
                           (g (make-id-mapper (quote-syntax (unbox imp^2))))
                           (x (make-id-mapper (quote-syntax (unbox imp^3)))))
                (letrec-syntaxes+values
                    (((m)
                      (syntax-rules ()
                        ((_ arg) (lambda () (g (+ y arg))))))
                    (((y)
                      (f x)))
                  ——— f ——— g ——— x ——— y ———
                  (m ———)
                  ———))))
          (hash-table))))))
```

Figure 6.2. Unit compilation with macros in signatures compile-time function creates a macro transformer that performs this replacement).

The code for definitions in the signature are stored in the signature's compile-time information, and copied into the unit's **letrec-syntaxes+values** expression.[1] The references in the definitions to imported variables are bound in the scope of the **let-syntax** transformers, so they will all be eventually replaced with the correct unbox expressions. Thus, the (f x) call becomes ((*unbox* *imp*^{1}) (*unbox* *imp*^{3})), and the (**m** ———) macro call becomes (**lambda** () (g (+ y ———))) which in turn expands to the following.

```
(lambda ()
  ((unbox imp²) (+ y ———)))
```

The code that is copied into the unit from the signature is stored as a syntax object in the signature's compile-time information, and as a syntax object its variables have scoping information attached. This scoping information is kept when the code is placed in the unit, so that the original bindings are referenced, and lexical scope is preserved. For example, in the following, the + reference in **m** inside of the unit will still refer to the + in scope at the signature's definition.

```
(define u
  (let ((+ 1)
        (f 2)
    (unit ———))))
```

Similarly, the f inside of the definition of y has the same scoping information as the signature's f, and it will always be bound to the **let-syntax**'s f (the f identifier used here comes from the signature's compile-time information, just as the definitions do).

When an imported signature has renaming or prefixing, the import names that are used in the unit differ from the names used inside of signature-contained definitions. In this circumstance, the unit macro emits extra macros that transform uses of the external (signature-specified) name into the internal (unit-import-specified) name.

6.3 Signature Extensions

A signature form definition, such as **open** or **struct**, is essentially a new kind of macro definition, created with **define-signature-form**. It is implemented as a macro-based extension to Scheme, and it uses the same technique as **define-signature** to communicate compile-time information: in this case, a signature macro is placed in a structure that

[1]**letrec-syntaxes+values** binds multiple macros and values in a single recursive scope.

marks it as a signature macro, and that structure is bound using **define-syntax** to the name of the signature form.

As the **define-signature** macro processes the body of the signature definition, it can encounter either variables (e.g., x), or applications (e.g., (m x)). When it encounters an application, it looks up m's definition in the compile-time environment (and raises a compilation error if there is none). It then applies the resulting function to the application's code, represented as a syntax object. The **define-signature** macro expects the result to be a list of syntax objects which it adds to the list of signature elements still to process.

Thus, the implementation of **define-signature** contains a simple macro expander for macro calls inside of signatures. Because scoping issues are the same for signature macros and the usual Scheme macros, syntax objects can represent the input and output of signature transformers. Thus, the complexity of lexically scoped macro expansion that is encapsulated in the syntax objects does not need to be reimplemented for signature macros.

6.4 First-Order Units

The unit definition forms (**define-unit**, **define-unit-binding**, etc.) use **define-syntax** to bind a compile-time description of the unit value's interface to the unit's name. The unit inference forms (e.g., **compound-unit/infer**) use the compile-time interface information as input to an inference algorithm that specifies how the units should be linked together. An inference form expands to a noninference form (e.g., **compound-unit**) form with the linkages fully specified.

6.5 Experience

The component examples of Chapters 4 and 5 all run on the implementation of units for Scheme described in this chapter. Furthermore, I have converted approximately 200 units in the DrScheme code base from the existing unit system to my new unit system. Many of the changes were trivial because the existing units did not represent component decompositions that would benefit from compile-time information. However, several signatures did include structure shape information, which was supported in the old unit system primitively, but is supported in my system as an extension to the core signature functionality.

In addition to the GUI framework (Section 5.3), the slideshow [Findler and Flatt 2004] collection[2] employs the structural matching pattern for creating a unit as a compounding of several units according to an internal decomposition, but whose signature from the perspective of clients of the library should be monolithic.

The vast majority of unit definitions were directly visible from the usage site, and each of the 15 converted instances of unit compounding used an inference form to do the compound. The **define-unit-binding** form was used only five times to set up these compounds, including once in the web server where it was used to give a compile-time interface to a unit-valued function parameter. Table 6.1 lists the sizes of the compound unit expressions before and after the conversion, along with the source file the compound resides in. Sizes are the number of symbols in the expression, not the number of lines, because of the flexibility inherent in laying out Scheme code.

Sixty-two unit definitions exhibited the following particularly simple idiom of creating a module simply to hold a unit.

```
(module name mzscheme
  (require (lib "unit.ss")
           ———)
  (provide name-unit)
  (define-unit name-unit (import ———) (export ———)
    ———))
```

I abstracted this pattern into a new language called *a-unit* that allows for a compact representation of the pattern.

```
(module name (lib "a-unit.ss")
  (require ———)
  (import ———)
  (export ———)
  ———)
```

A similar language, *a-signature*, supports the analogous abstraction for signature definitions.

[2]actually the *texpict* collection that slideshow relies on.

Table 6.1. Compound unit sizes

File	# of syms before	# of syms after	before / after
web-server/configuration.ss	56	69	123%
web-server/launch.ss	23	20	87%
web-server/web-server-unit.ss	27	10	37%
web-server/web-server.ss	26	15	58%
texpict/mrpict-unit.ss	36	10	28%
texpict/texpict-unit.ss	32	9	28%
test-suite/tool.ss	38	12	32%
stepper/stepper+xml-tool.ss	23	13	57%
profjBoxes/tool.ss	38	12	32%
help/private/link.ss	81	56	69%
handin-server/web-status-server.ss	47	44	94%
graphics/graphics-unit.ss	29	10	34%
framework/framework-unit.ss	291	149	51%
drscheme/private/link.ss	195	27	14%
browser/browser-unit.ss	80	18	22%

APPENDIX

UNIT GRAMMAR

The syntax of the unit system is introduced in Figures 4.4, 4.6, 4.10, 4.11, 5.2, 5.3, 5.4, and 5.6. I collect here the complete syntax, including an additional *tag* feature. Tags are useful when a unit wants to import the same signature multiple times (including sub-signatures) which usually causes a compile-time error due to ambiguity. A tag on a signature becomes part of the unit's interface and allows the unit's user to specify linkages for particular interfaces.

Each of the following nonterminals refers to Scheme symbols (I use the different names to clarify the intended use of the symbol): *link-id*, *sig-id*, *id*, *unit-id*, *value-id*, *macro-id*, *sig-form-id*, and *struct-id*. The *expr* and *def* nonterminals are for Scheme expressions and definitions. I only give here the unit-related productions. The *s-expr* nonterminal is for an arbitrary piece of Scheme syntax. The *nonterminal** notation indicates that the nonterminal can occur zero or more times.

I use the following abbreviation to concisely convey the optional tagging mentioned above. When $\mathcal{T}[x]$ is used as a nonterminal, it indicates a reference to a new nonterminal defined as follows.

x | (**tag** *id* x)

Thus, $\mathcal{T}[x]$ refers to x with an optional tag added.

unit-expr = (**unit** (**import** $\mathcal{T}$[*sig-expr*]*) (**export** $\mathcal{T}$[*sig-expr*]*) *init-dep def**)
| (**compound-unit** (**import** $\mathcal{T}$[*link-spec*]*) (**export** $\mathcal{T}$[*link-id*]*)
(**link** *linkage**)
| (**compound-unit/infer**
(**import** *link-spec/infer**)
(**export** *export-spec/infer**)
(**link** *linkage/infer**))
| (**unit-from-context** $\mathcal{T}$[*sig-expr*])
| (**unit/new-import-export**
(**import** $\mathcal{T}$[*sig-expr*]*) (**export** $\mathcal{T}$[*sig-expr*]*) *init-dep*
(($\mathcal{T}$[*sig-expr*]*) *expr* $\mathcal{T}$[*sig-expr*]*))
sig-expr = *sig-id*
| (**prefix** *symbol sig-id*)
| (**rename** *sig-id ren**)
ren = (*id id*)
init-dep = ϵ
| (**init-depend** $\mathcal{T}$[*sig-id*]*)
link-spec = (*link-id* **:** *sig-id*)
linkage = (($\mathcal{T}$[*link-spec*]*) *expr* $\mathcal{T}$[*link-id*]*)
link-spec/infer = *sig-id*
| *link-spec*
export-spec/infer = *link-id*
| *sig-id*
linkage/infer = *unit-id*
| (($\mathcal{T}$[*link-spec*]*) *unit-id* $\mathcal{T}$[*link-id*]*)
sig-def = (**define-signature** *sig-id* (*spec**))
| (**define-signature** *sig-id* **extends** *sig-id* (*spec**))
spec = *value-id*
| (**define-syntaxes** (*macro-id**) *expr*)
| (**define-values** (*value-id**) *expr*)
| (*sig-form-id s-expr**)
| (**open** *sig-expr*)
| (**struct** *struct-id* (*id**) *omit**)
omit = ϵ
| **-type**
| **-selectors**
| **-setters**
| **-constructor**

def = (**define-values/invoke-unit** *expr* $\mathcal{T}$[*sig-expr*]*)
| (**define-values/invoke-unit/infer** *expr*)
| (**define-values/invoke-unit/use-context** *expr*
  (**import** $\mathcal{T}$[*sig-expr*]*) (**export** $\mathcal{T}$[*sig-expr*]*))
| *sig-def*
| (**define-signature-form** *sig-form-id* *expr*)
| (**provide-signature-elements** *sig-expr**)
| (**define-unit** *unit-id* (**import** $\mathcal{T}$[*sig-expr*]*) (**export** $\mathcal{T}$[*sig-expr*]*) *init-dep*
  def*)
| (**define-unit-binding** *unit-id* *expr*
  (**import** $\mathcal{T}$[*sig-id*]*) (**export** $\mathcal{T}$[*sig-id*]*) *init-dep*)
| (**define-compound-unit** *unit-id* (**import** $\mathcal{T}$[*link-spec*]*) (**export** $\mathcal{T}$[*link-id*]*)
  (**link** *linkage**))
| (**define-compound-unit/infer** *unit-id*
  (**import** *link-spec/infer**)
  (**export** *export-spec/infer**)
  (**link** *linkage/infer**))
| (**define-unit/new-import-export** *unit-id*
  (**import** $\mathcal{T}$[*sig-expr*]*) (**export** $\mathcal{T}$[*sig-expr*]*)
  (($\mathcal{T}$[*sig-expr*]*) *expr* $\mathcal{T}$[*sig-expr*]*))
| (**define-unit-from-context** *unit-id* $\mathcal{T}$[*sig-expr*])
⋮

expr = (**invoke-unit** *expr*)
| *unit-expr*
⋮

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, USA.

Ancona, D., Damiani, F., Drossopoulou, S., and Zucca, E. 2005. Polymorphic bytecode: Compositional compilation for Java-like languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 26–37.

Ancona, D. and Zucca, E. 2001. True modules for Java-like languages. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*. Springer-Verlag, Berlin, Germany, 354–380.

Ancona, D. and Zucca, E. 2002. A calculus of module systems. *Journal of Functional Programming 12,* 2, 91–132.

Batory, D., Sarvela, J. N., and Rauschmayer, A. 2004. Scaling step-wise refinement. *IEEE Transactions on Software Engineering 30,* 6, 355–371.

Bawden, A. 2000. First-class macros have types. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 133–141.

Blume, M. and Appel, A. W. 1999. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems 21,* 4, 813–847.

Boudol, G. 2004. The recursive record semantics of objects revisited. *Journal of Functional Programming 14,* 3, 263–315.

Cejtin, H., Fluet, M., Jagannathan, S., and Weeks, S. 2005. Formal specification of the ML basis system. `http://mlton.org/pages/MLBasis/attachments/mlb-formal.pdf`.

Clinger, W. and Rees, J. 1990. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 155–162.

Crary, K., Harper, R., and Puri, S. 1999. What is a recursive module? In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 50–63.

Culpepper, R., Owens, S., and Flatt, M. 2005. Syntactic abstraction in component interfaces. In *GPCE '05: 4th International Conference on Generative Programming and Component Engineering*. Springer, Berlin, Germany.

DeRemer, F. and Kron, H. 1975. Programming-in-the-large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software*. ACM Press, New York, NY, USA.

Dreyer, D. 2004. A type system for well-founded recursion. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, New York, NY, USA, 293–305.

Dreyer, D. 2005a. Recursive type generativity. In *ICFP '05: Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming.* ACM Press, New York, NY, USA, 41–53.

Dreyer, D. 2005b. Understanding and evolving the ML module system. Ph.D. thesis, Carnegie Mellon University.

Dreyer, D., Crary, K., and Harper, R. 2003. A type system for higher-order modules. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, New York, NY, USA, 236–249.

Duggan, D. and Sourelis, C. 1996. Mixin modules. In *ICFP '96: Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming.* ACM Press, New York, NY, USA, 262–273.

Dybvig, R. K., Hieb, R., and Bruggeman, C. 1992. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation 5,* 4, 295–326.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M. 2002. DrScheme: A programming environment for Scheme. *Journal of Functional Programming 12,* 2, 159–182.

Findler, R. B. and Flatt, M. 1998. Modular object-oriented programming with units and mixins. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming.* ACM Press, New York, NY, USA, 94–104.

Findler, R. B. and Flatt, M. 2004. Slideshow: Functional presentations. In *ICFP '04: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming.* ACM Press, New York, NY, USA, 224–235.

Findler, R. B. and Flatt, M. 2006. *PLT Framework: GUI Application Framework*, 350 ed. `http://download.plt-scheme.org/doc/350/html/framework/`.

Flatt, M. 2002. Composable and compilable macros: You want it when? In *ICFP '02: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming.* ACM Press, New York, NY, USA, 72–83.

Flatt, M. 2006. *PLT MzScheme: Language Manual*, 350 ed. `http://download.plt-scheme.org/doc/350/html/mzscheme/`.

Flatt, M. and Felleisen, M. 1998. Units: Cool modules for HOT languages. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation.* ACM Press, New York, NY, USA, 236–248.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2000. *The Java Language Specification, second edition.* Addison-Wesley, Boston, USA.

HARPER, R. AND LILLIBRIDGE, M. 1994. A type-theoretic approach to higher-order modules with sharing. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 123–137.

HIRSCHOWITZ, T. AND LEROY, X. 2005. Mixin modules in a call-by-value setting. *ACM Transactions on Programming Languages and Systems 27,* 5, 857–881.

HIRSCHOWITZ, T., LEROY, X., AND WELLS, J. B. 2003. Compilation of extended recursion in call-by-value functional languages. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM Press, New York, NY, USA, 160–171.

HIRSCHOWITZ, T., LEROY, X., AND WELLS, J. B. 2004. Call-by-value mixin modules: Reduction semantics, side effects, types. In *ESOP 2004: Proceedings of the 13th European Symposium on Programming*. Springer, Berlin, Germany, 49–63.

JONES, S. P., Ed. 2003. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK.

KELSEY, R., CLINGER, W., AND REES (EDITORS), J. 1998. Revised^5 report of the algorithmic language Scheme. *ACM SIGPLAN Notices 33,* 9, 26–76.

KELSEY, R., REES, J., AND SPERBER, M. 2005. *Scheme48 Reference Manual*, 1.3 ed. `http://s48.org/1.3/s48manual.pdf`.

KOHLBECKER, E. E., FRIEDMAN, D. P., FELLEISEN, M., AND DUBA, B. F. 1986. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM Press, New York, NY, USA, 151–161.

KRISHNAMURTHI, S. 2001. Linguistic reuse. Ph.D. thesis, Rice University.

LEROY, X. 1994. Manifest types, modules, and separate compilation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 109–122.

LEROY, X. 1995. Applicative functors and fully transparent higher-order modules. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 142–153.

LEROY, X. 1996. A syntactic theory of type generativity and sharing. *Journal of Functional Programming 6,* 5, 667–698.

LEROY, X. 2003. A proposal for recursive modules in Objective Caml. `http://caml.inria.fr/pub/papers/xleroy-recursive_modules-03.pdf`.

LEROY, X. 2004. *The Objective Caml System*, 3.08 ed. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

LOPEZ-HERREJON, R. E., BATORY, D., AND COOK, W. 2005. Evaluating support for features in advanced modularization technologies. In *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*. Springer-Verlag, Berlin, Germany.

Makholm, H. and Wells, J. B. 2005. Type inference, principal typings, and let-polymorphism for first-class mixin modules. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming.* ACM Press, New York, NY, USA, 156–167.

McDirmid, S., Flatt, M., and Hsieh, W. C. 2001. Jiazzi: New-age components for old-fasioned Java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.* ACM Press, New York, NY, USA, 211–222.

McIlroy, M. D. 1969. "Mass produced" software components. In *Software Engineering*, P. Naur and B. Randell, Eds. Scientific Affairs Division, NATO, Brussels, 138–155. Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. 1997. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, Massachusetts, USA.

Odersky, M. and Zenger, M. 2005. Scalable component abstractions. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.* ACM Press, New York, NY, USA, 41–57.

Okasaki, C. 1998. *Purely Functional Data Structures.* Cambridge University Press, Cambridge, UK.

Pierce, B. C. 2002. *Types and Programming Languages.* The MIT Press, Cambridge, Massachusetts, USA.

PLT. 2006a. *PLT Miscellaneous Libraries: Reference Manual*, 350 ed. `http://download.plt-scheme.org/doc/350/html/misclib/`.

PLT. 2006b. *PLT MzLib: Libraries Manual*, 350 ed. `http://download.plt-scheme.org/doc/350/html/mzlib/`.

Russo, C. V. 2001. Recursive structures for Standard ML. In *ICFP '01: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming.* ACM Press, New York, NY, USA, 50–61.

Serrano, M. 2004. *Bigloo: A "practical Scheme compiler"*, 2.6e ed. `http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html`.

Swasey, D., Murphy VII, T., Crary, K., and Harper, R. 2006. A separate compilation extension to Standard ML (working draft). Tech. Rep. CMU-CS-06-104, School of Computer Science, Carnegie Mellon University. January.

Syme, D. 2006. Initializing mutually referential abstract objects: The value recursion challenge. *Electronic Notes in Theoretical Computer Science 148,* 2, 3–25. Proceedings of the 2005 ACM SIGPLAN Workshop on ML.

Szyperski, C. 1997. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, New York.

Vinoski, S. 1997. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine 35,* 2, 46–55.

Waddell, O. and Dybvig, R. K. 1999. Extending the scope of syntactic abstraction. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, New York, NY, USA, 203–215.

Wadler, P. 1987. Views: A way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* ACM Press, New York, NY, USA, 307–313.

Wells, J. B. and Vestergaard, R. 2000. Equational reasoning for linking with first-class primitive modules. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems.* Springer-Verlag, Berlin, Germany.

Wirth, N. 1982. *Programming in Modula-2.* Springer-Verlag, Berlin, Germany.

Wright, A. K. and Felleisen, M. 1994. A syntactic approach to type soundness. *Information and Computation 115,* 1, 38–94.